Detection and Segmentation of QR codes with arbitrary deformations

Hamid Latif Martínez

30/06/2020

Supervisor: Javier Ruiz-Hidalgo
(Department of Signal Theory and Communications)

Tutor: Oscar Romero Moral
(Department of Service and Information System Engineering)

Cosupervisor: Ismael Benito Altamirano
(ColorSensing)

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
DATA SCIENCE
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSTITAT POLITÈCNICA DE CATALUNYA (UPC) - BarcelonaTech

Contents

Chapter 1

Introduction

1.1 Motivation

ColorSensing[1] is a *Universitat de Barcelona* spin-off that offers products for color correction by means of a QR (Quick Response) code in which color patterns are included (see figure 1.1). These QR codes are used by including them in the scene of which the user wants to take a picture. By having that QR code in the image and detecting it, the pattern colors that it includes can be used by the algorithm to correct the color shift of the image to generate a color palette as similar to the original one, and thus correcting the error generated by the camera.

Figure 1.1: *ColorSensing's* custom QR codes.

Currently, the detection of QR codes is achieved by means of an internal decoder that uses classical computer vision algorithms. This works pretty well for most cases, but can be prone to some problems whenever the QR code has certain arbitrary deformations. For instance, whenever the printed QR code is put on a bottle, which deforms the QR with a cylindrical deformation.

These problems can be partially mitigated by introducing algorithms able of detecting and correcting the most common deformations expected (cylindrical, perspective, etc.). Even though this approach can be used to handle the main expected deformations, is inappropriate to handle arbitrary deformations.

To avoid these problems and to try to obtain a better result, using a neural network to try to

detect the QR codes seems worth exploring.

Considering the tools available at *ColorSensing*, three main tasks could be of help:

1. **Object detection**: Being able of detecting the QR in an image (i.e. generating a bounding box in which the code is located), as the sub-image contained by the bounding box could be fed to the internal decoder. If the decoder also struggles with this image, it can be stored to manually study why it failed (and thus bring value).
2. **Version classification**: If the neural network is able to detect the QR version (see section 1.2), this could be of help for the decoder, as the version changes the QR code shape.
3. **Instance segmentation**: Segmenting the QR code (i.e. producing an image mask indicating exactly at which pixels the code is located) can be pretty useful to know if the code is deformed and, in that case, the deformation type. This could be used by the decoder to correct it.

These three main tasks are further explained in section 1.5, and are the main motivation why a neural network is used.

1.2 What are QR Codes?

The QR Code (*Quick Response*) was invented in the 1994 by the Japanese automotive industry. According to the creators of the QR code[2], it *"is a matrix code that belongs to a larger set of machine-readable codes, all of which are often referred to as barcodes, regardless of whether they are made up of bars, squares or other-shaped elements"*.

Its purpose is to have a quick and efficient way of encoding small amounts of information (depending on the QR Code version, it may contain up to a total of 7.089 numerals) in a format that is easy and fast to decode. The massification of smartphones has increased its popularity.

Furthermore, QR Codes have advanced error-correction methods and characteristics that allow it to be reader reliably, as the errors that it may contain can be corrected (depending on the error-correction level, up to a 30% of the error may be corrected.)

QR Codes have three components that are present no matter what version it is: a *Position detection pattern*, an *Alignment pattern*, a *Timing pattern* (see figure 1.2).

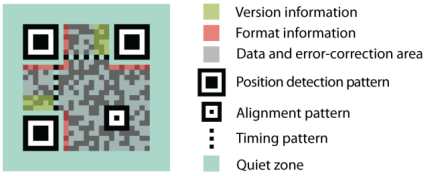


Figure 1.2: Structure of a QR code version 2. (Source: [2])

1.2.1 QR code Versions

There are 40 different versions of QR codes, ranging from 1 to 40. Codes with version 1 have a size of 21x21 modules, codes with version 2 have 25x25 modules, and so on. The biggest codes, therefore, are QR codes with version 40, which have a size of 177x177 modules.

The size of the data that the QR code can contain also increases with the version.

In addition to the difference in size, the different QR code versions also change the elements that the code contains. These elements are exposed in sections 1.2.2, 1.2.3, 1.2.4 and 1.2.5.

1.2.2 Position Detection Pattern

The Position Detection Patterns are located in three corners of a QR code, and their main purpose is to help localize the codes. Since these patterns have a symmetrical scan-line ratio of 1:1:3:1:1 (see figure 1.3), they are easily scanned from any direction within 360º (see figure 1.4). Furthermore, their positional relationship allows an easy way of calculating the relevant angle, position and size information contained in the code.

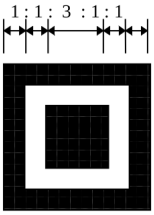


Figure 1.3: Structure of a Position Detection Pattern. (Source: [3])

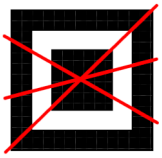


Figure 1.4: Position Detection Patterns can be scanned from any angle.

1.2.3 Alignment Pattern

The Alignment Patterns are fixed reference patterns in defined positions of the QR code. They enable the decoding software to re-synchronize the coordinate mapping of the image modules when moderate amounts of deformations affect the code. The number of Alignment Patterns depends on the QR code version, and are located in codes with versions 2 and higher.

1.2.4 Timing Pattern

The Timing Patterns are located horizontally and vertically (as seen in figure 1.2), and consist of alternating dark and light modules. It always starts and ends with a dark module. Timing patterns enable both density and version to be determined and helps determine module coordinates.

1.2.5 Quiet Zone and Encoding Region

The Encoding Region is the region where all the data is encoded. On the other hand, the Quiet Zone is a region that must be free of any data, and must surround the QR code on all four sides. Its value should be equal to the light modules (seen in section 1.2.4.

1.2.6 Error correction

As stated in section 1.2, QR codes employ error correction codewords. These are added to the data to enable correction of loss in data. There are four levels of error correction, shown table 1.1.

Error correction level	**Recovery capacity (%)**
L	7
M	15
Q	25
H	30

Table 1.1: Error correction levels.

1.2.7 The use of QR codes in *ColorSensing*

The characteristics of QR codes explained above make them perfect for the product developed at *ColorSensing*, which is a label encoding colors to be used in the color correction process.

1.3 A brief history of Machine Learning applied to Computer Vision

Computer vision has been a very prolific field in the last years, but its origins can be pinpointed in the fifties. In 1959, Huber and Wiesel [4] studied how several hundred units in the cat's striate cortex react to visual stimuli. Their idea was to show the cats different patterns of light to see how the cortex neurons reacted to them to try to "debug" how it worked.

Through these experiments, the researchers were able to conclude that the visual processing always starts with simple structures such as oriented edges. Keep this idea in mind, for it will be at the core of the models about to be explained.

In 1963 Lawrence Robert's PhD thesis, named *"Machine perception of three-dimensional solids"* [5], described the process of deriving 3D information about objects from 2D images. This idea would be one of the precursors of the latter Computer Vision advances.

In his thesis, Lawrence Robert described how to process 2D images as draw lines and then using these to create 3D representations and, finally, display these representations as 3D structures in which the hidden lines were removed.

During these years, Seymour Papert though that the problem of machine vision could be solved in a few months and, to that end, launched the *Summer Vision Project* [6]. Papert, a professor at MIT (Massachusetts Institute of Technology), though that a group of MIT students would be able to successfully tackle the problem and have it solved short after. But this project would not have existed if he and his students succeeded, so it may come without saying that they did not.

In 1982 David Marr published the paper *"Vision: A computational investigation into the human representation and processing of visual information"* in which he established a new concept: vision is hierarchical. He proposed that the main function of the vision system is to create 3D representations of the environment, so that it can be interacted with. He also introduced a framework for vision in which vision is divided into two parts: low-level algorithms to detect corners, edges, etc. and a high-level understanding of the data, which as deduced out of the low-level algorithms data.

At around the same time, in 1980 Fukushima published *"Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position"* [7], in which he proposed a neural network model for a mechanism of visual pattern recognition. The network, according to the paper, *"is self-organized by learning without a teacher, and acquires an ability to recognize stimulus patterns based on the geometrical similarity of their shapes"*. This network is the first that resembles the Convolutional Neural Networks that are used currently (see figure 1.5). Back then, though, this network structure was not widely used due to the limits in computation hardware needed to train it. [8]

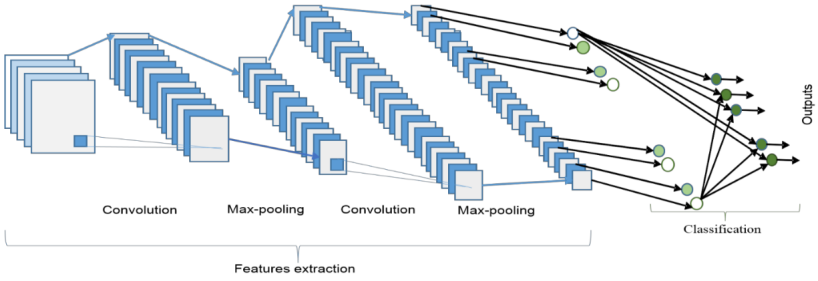


Figure 1.5: Architecture of a Convolutional Neural Network. (Source: [8])

As seen in figure 1.5, the architecture of a CNN consists of two main parts: feature extractors and a classifier. A CNN is a combination of layers, which can be of the following types: convolution, pooling and classification.

The convolution layers get their name out of the convolution operation. This operation consists in the multiplication of the input matrix (the image) by an nxn matrix (which is the filter, where n is an odd number, usually 1, 3 or 5) in a sliding window fashion. The convolution operation consists of the following steps (see figure 1.6):

1. Place the filter at the beginning of the input matrix, beginning from the top-left corner.

2. Perform an element-wise multiplication of the image matching cells (i.e. the image cells that overlap with the filter) and the filter. Store the sum of these values in the first column of the first row of the result matrix.

3. Move the filter one position to the right, and repeat.

4. When the entire row has been processed, move one column down and repeat this process from the first column.

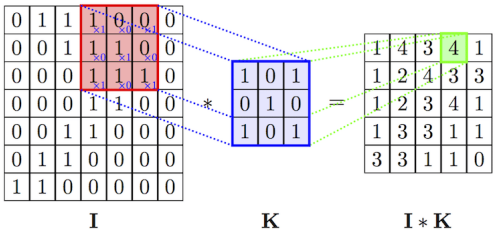


Figure 1.6: An example of a convolution operation.

The pooling layers receive their name because of the pooling operation. An example of a pooling operation may be max-pooling. The max-pooling operation consists of the following steps (see figure 1.7):

1. Place the filter at the beginning of the input matrix, beginning from the top-left corner.

2. Select the element with the maximum value, and store it in the first column of the first row of the result matrix.

3. Move the filter one position to the right, and repeat.

4. When the entire row has been processed, move one column down and repeat this process from the first column.

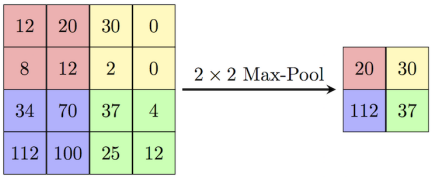


Figure 1.7: An example of a max-pooling operation.

Lastly, the classification layers are located at the end and their task is (as its name tells) perform classifications with the features received from the feature extractor.

Each of the feature extraction layers receive the output from its immediate predecessor layer, and passes its output as input for the next one. The layers that form the feature extractor are of two possible types: convolution and pooling. They are combined in the following way: the even numbered layers are for convolutions, whereas the odd numbered layers are for max-pooling.

Therefore, the main idea of CNNs is that low-level features are combined to create higher-level features, which are then used for classification. This idea is, clearly, derived from the frameworks and algorithms presented in the research published by David Marr, Lawrence Roberts, and many others.

In 1998, LeCun et. al [9] proposed *LeNet*, a network very similar to Fukushima's, but applying back-propagation [10] to train it end-to-end using supervised learning.

Back-propagation is an algorithm to compute the gradient of the cost function. The full description of the algorithm is the following:

Data: A network with l layers, the activation function σ_l, the outputs of hidden layer $h_l = \sigma_l(W_l^T h_{l-1} + b_l)$, the network output $\widetilde{y} = h_l$, the layer weights W_l and the unit bias b_l

Result: The accumulated gradient of all the layers: $\delta \leftarrow \frac{\partial \epsilon(y_i, \widetilde{y_i})}{\partial y}$

for $i \leftarrow l$ *to* 0 **do**

Calculate gradient for present layer:

$\frac{\partial \epsilon(y,\widetilde{y})}{\partial W_l} = \frac{\partial \epsilon(y,\widetilde{y})}{\partial h_l} \frac{\partial h_l}{\partial W_l} = \frac{\partial h_l}{\partial W_l}$

$\frac{\partial \epsilon(y,\widetilde{y})}{\partial b_l} = \frac{\partial \epsilon(y,\widetilde{y})}{\partial h_l} \frac{\partial h_l}{\partial b_l} = \frac{\partial h_l}{\partial b_l}$

Apply gradient descent using $\frac{\partial \epsilon(y,\widetilde{y})}{\partial W_l}$ and $\frac{\partial \epsilon(y,\widetilde{y})}{\partial b_l}$

Back-propagate gradient to the lower layer:

$\delta \leftarrow \frac{\partial \epsilon(y,\widetilde{y})}{\partial h_l} \frac{\partial h_l}{\partial h_{l-1}} = \delta \frac{\partial h_l}{\partial h_{l-1}}$

end

Algorithm 1: Backpropagation algorithm.

As can be seen in algorithm 1, provided an efficient way of training a whole network. By using it, *LeNet* achieved an impressive performance on handwritten digits dataset.

The activation functions used in the backpropagation algorithm (see algorithm 1) are what define the output of a unit given a set of inputs. The simplest activation neuron is a binary one, in which the unit is either "active" or "inactive" (i.e. it outputs a 1 or a 0). Choosing the correct activation functions is very important, as they can cause the network to train poorly as studied in [11].

Later on, in 2012, Alex Krizhevesky et. al proposed *AlexNet* [12] a deeper and wider CNN model. With this new model, the authors were able to win the *ImageNet* challenge for object recognition: the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

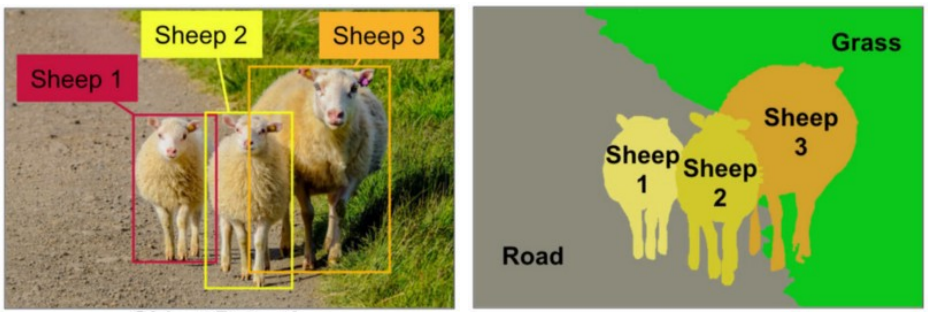
The success achieved with *AlexNet* increased the interest in deep learning, and from that moment on, the performance of the successive networks that were created augmented rapidly, which helped in the progress of the Computer Vision field.

Many newer models are available nowadays, and some of those are described in section 2.

1.4 Context

Most of the QR Code detectors and decoders publicly available today are based on classic computer vision algorithms. Some of these examples are *ZXing* [13] or *pyzbar* [14].

In this thesis, object detection and instance segmentation are discussed. The differences between each other are that in object detection, a bounding box containing the detected object is generated, whereas in instance segmentation a mask containing every pixel forming the detected object is created. See figure 1.8 for a visual example.



(a) (b)

Figure 1.8: (a) Object detection (b) Instance segmentation (Source: [15])

There are some studies published regarding the use of neural networks to detect and locate QR Codes, such as [16] or [17], but the authors do not distribute the trained model to use it as a locator. Therefore, its use as an installable detector is not possible without manually downloading the code, the datasets and training the model.

To the knowledge of the author, there are no publications regarding the use of neural networks to perform instance segmentation with QR codes.

1.5 Objectives

As the project's title states, the main objectives will be to *detect* and *segment* QR codes with arbitrary deformations, in that order. The objectives have been clearly separated into two because even though they seemed similar in the beginning, each one has its peculiar strengths and weaknesses. Developing models that are able to perform each one of them and studying how they perform is vital to understand if they can be applicable in an industrial environment.

Specifically, the expected differences between the two objectives that justify their split are the following:

1. Detecting an object is faster than segmenting it. State of the art object detection models, such as *YOLO*, can make inferences in up to 125 images per second[18], which is faster than the newest models able to segment images [19]. Since one of the main motivations of this project

is to achieve a model as fast as possible, it seems wise to try a detection and segmentation models separately.

2. As explained in previous sections, the QR Code present in the image may be subject to arbitrary deformations, drawing a bounding box around it may be unusable, as the decoder may not be able to decode it without knowledge of the deformations. On the other hand, having a mask may help the decoder in its task.

3. An object detection model needs a data set with information about the bounding boxes where the ground truth is located at, whereas a segmentation model needs the mask. Since there are publicly available decoders able of generating a bounding box around a QR Code, but not decoders that build the complete mask, it may be easier to automatically generate a big dataset for the first model. On the other hand, the latter may need a manually generated data set, which takes a huge amount of time.

1.6 Structure of this report

The chapters in this master thesis are structured in a way that tries to help the reader follow the story of this project: it starts by explaining the issues that motivated this work and it them moves on to explain the current state in this very specific task. Right after that, a theoretical explanation of the architecture used is explained, which is followed by the chapters explaining the data preprocessing steps and the most interesting aspects of the implementation. The experiments comes next, which try to answer the main raised hypotheses detailed at the beginning of the chapter. After that, conclusions are discussed and the future work is explained. The thesis ends with an acknowledgements section.

Chapter 1 gives a general overview of the motivations that led to this project. It provides a general overview of the issues that *ColorSensing* had in detecting QR codes with arbitrary deformations, it explains the main aspects of QR codes and explains the main objectives of this project.

Chapter 2 discusses the state of the art in terms of object detectors and explains one very popular instance segmentation model. This section is divided into two main parts: the first one briefly discusses the classical methods for object detection, whereas the second one explains methods based on neural networks. This chapter is closed with an example of how *YOLO* is used to perform QR codes detection, mainly because *YOLO* is the chosen architecture to perform that task in this project. This serves, therefore, to give the reader an overall idea of the existing previous work at the moment of starting this project.

Chapter 3 gives a deeper theoretical explanation of how *YOLO*, which serves as a way to help the reader focus on the architecture used in this project.

Chapter 4 explains the proposed solution, with explanations of the first decisions taken.

Chapter 5 details the data preprocessing, which spans all the work that had to be done before any experiment could be made.

Chapter 6 gives some details of the more interesting bits of the implementation part. Both this and the previous chapters try to help the reader reproduce the experiments.

Chapter 7 is the most interesting one, as it details the hypotheses and the experiments that were performed to confirm or reject them. Every experiment has the same three-level structure: the first one describes the settings used to perform the experiments, the second one details the results of the experiments and the third discusses the observed results.

Chapter 8 provides the main conclusions of the experiments performed for this project, and discusses the most interesting parts.

Chapter 9 gives some further experiments and ideas that the author would like to test in the future. These have been a product of the results and its interpretations.

Chapter 10 serves to show appreciation to all the people who helped in the realization of this project.

Chapter 2

State of the art

In this section, the state of the art of object detectors that could be applied to detect QR codes are explained, both those that are based in classical methods and those that are based on neural networks.

2.1 Detectors based on classical methods

An example of how classical methods work to detect QR codes can be found in [20]. In this paper, the authors explain how the use of Viola-Jones detection framework to find QR codes in images.

2.1.1 Viola-Jones framework

Viola-Jones framework works in three steps:

1. **Haar-like Features**: Use feature prototypes (proposed in [21]) known as Haar-like features. The feature values are computed with respect to a certain sample, which is a sub-region of the image under a sliding window.
2. **Integral image**: The integral image is a matrix with a shape equal to the input image. Every *(x, y)* position of the integral image contains the sum of all pixel values in the rectangle between *(0, 0)* and *(x, y)*.
3. **Classifier Cascade and Boosting**: A cascade, which consists in a series of consecutive stages whose task is to reject samples that are not equal to the pattern being searched.

The proposed method consists of two main steps, which are the detection of FIP candidates (which is how the authors refer to the Position detection patterns; see figure 1.2) using the Viola-Jones framework, and then post-process the candidates to determine the size and position of the QR code.

The FIP candidates are used because the authors explain that *"in Viola-Jones framework it is assumed that the pattern to be detected should possess a rigid structure, but no strong restrictions concerning pattern size is assumed."*. Knowing that FIP patterns are present in all QR codes (as can be seen in figure 1.2), and that their structure is rigid, they are excellent targets to be detected with the Viola-Jones framework, which is the reason why the authors use them.

2.2 Detectors based on neural networks

2.2.1 R-CNN

Moving on to neural network based approaches, a very popular approach are *R-CNN* (Region-based CNNs). *R-CNN* models start by selecting several regions from an image (using region proposal methods, as explained later), use a CNN to extract features from each proposed area and then use these features to predict the categories and bounding boxes (see figure 2.1).

The four main components of *R-CNN* models are:

1. Several high interest regions are proposed with selective search. These regions are usually selected on multiple scales and may have different shapes.
2. A pretrained CNN transforms each of these proposed regions into the required network input and extract features from them.
3. The features and labeled categories extracted from the previous steps are used by multiple *SVMs* (Support Vector Machines) for object classification. Each of those *SVMs* determines whether the object belongs to a certain category or not.
4. The features and bounding boxes of each interest region are combined to train a linear regressor model to perform ground-truth bounding box predictions.

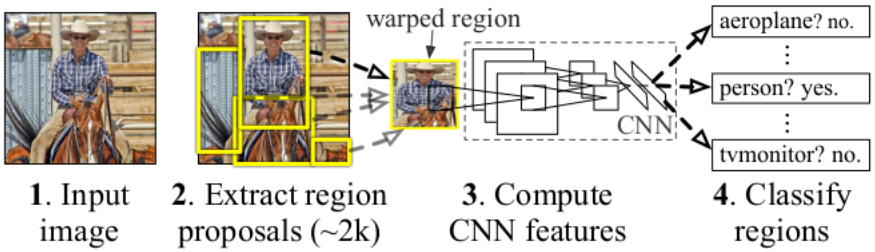


Figure 2.1: R-CNN architecture. (Source: [22])

These multiple steps, as well as the fact that the each image is checked multiple times, make *R-CNNs* very slow. *Fast R-CNN* originated to try to solve this.

2.2.2 Fast R-CNN

The main changes introduced by *Fast R-CNN* are:

1. In contrast with *R-CNNs*, here the entire image is sent to a CNN to perform feature extraction.
2. n regions with different shapes are proposed. Features of the same shapes must be extracted, and this is performed by RoI Pooling (Region of Interest), introduced in this new version. RoI Pooling uses the CNN output and interest regions to produce a tensor of the features extracted from each proposed region.
3. A fully connected layer transforms the output generated in the previous step to a shape suitable to the model design.

4. Now it is time for category prediction, which is performed by softmax regression to output a nxq tensor (so q are the number of categories). For the bounding box prediction, a tensor of size nx4 is produced.

2.2.3 Faster R-CNN

Successive changes were applied in *Faster R-CNN* [23]. These changes mainly affected the region proposal method. The introduced region proposal method is the following:

1. A 3x3 convolutional layer with padding 1 is used to transform the CNN output and set the number of output channels to c.
2. Each element in the feature map is used as center to generate multiple anchor boxes of different sizes and aspect ratios. Each anchor box is then labeled.
3. The features of the elements at the center of the anchor boxes are used to predict the binary category (either object or background) and their bounding box.
4. NMS (Non-Maximum Supression) is used to remove similar bounding boxes whose category has been predicted as object. The remaining bounding boxes after this process are proposed as regions of interest.

2.2.4 Mask R-CNN

Mask R-CNN is an iteration based on *Faster R-CNN* that introduced an extra mask head. This allowed segmentation at a pixel-wise level, which allows to identify every object separately.

The main changes introduced in this version with respect to *Faster R-CNN* were (see figure 2.3:

1. Use of FPNs [24] (Feature Pyramid Networks).
2. Additional branch for masks generation.
3. Replacement of RoIPool with RoIAlign, which does not digitalize the boundary of the cells and makes every target cell have the same size.

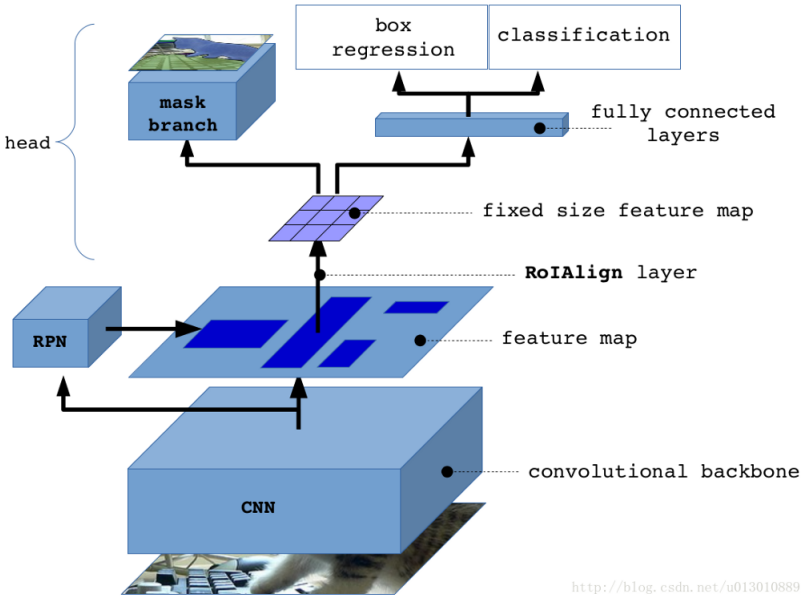


Figure 2.2: Mask R-CNN architecture. (Source: [25])

2.2.5 SSD

SSD (Single Shot Detector) is designed to perform object detection in real-time. The previous detectors (sections 2.1, 2.2.2, 2.2.3) have a a common problem: the speed at which these perform detections is quite slow (the fastest one, *Faster R-CNN*, is able to perform detections at 7 frames per second [26]).

SSD is able to speed up the process by eliminating the need of the region proposal network. The loss in accuracy suffered from this is compensated with some improvements, that include multi-scale features and default boxes, which allow *SSD* to match *Faster R-CNN's* accuracy using lower resolution images, which allows for a faster processing speed.

Since *YOLO* is also a single-shot detector model, no more details will be explained regarding *SSD*.

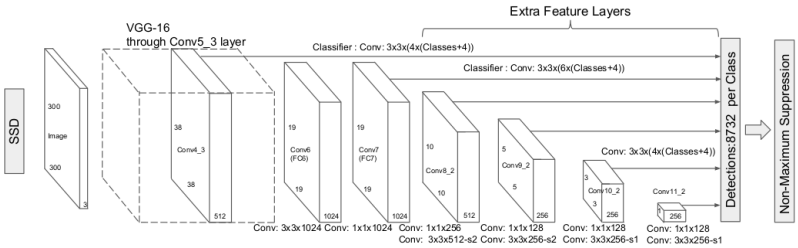


Figure 2.3: SSD architecture. (Source: [26])

2.2.6 YOLO: a specific use

YOLO (You Only Look Once) is a single-shot CNN to perform object detections (see chapter 3 for a deep theoretical explanation regarding how *YOLO* works).

Since *YOLO* has been the model chosen to perform detections (see chapter 4), this subsection will explain how previous publications have studied its use in the detection of QR codes.

To the knowledge of the author, the most relevant paper in which the use of *YOLO* to QR codes detection is [27]. In this paper, the authors describe their proposal, which consists of using *YOLO* based on *Darknet 19* (one of the versions of *YOLO*, see section 7.9 for an experiment in which it is used) to perform detections and angle predictions with an image input size of 416x416 (the networks used for object detection and angle prediction are different). The authors introduce a little modification in the softmax layer.

It must be noted that the authors describe that the network is trained to detect both 1D barcodes and QR codes, and therefore, the results will probably be different to the ones that could have been obtained if trained only to perform detections of QR codes.

The complete proposal is the following:

1. Using a *YOLO* network with input images of size 416x416, perform detections of QR codes and 1D barcodes.
2. Crop the images contained in the generated bounding boxes, square them and send them to the angle detector.
3. The angle detector, which is a *YOLO* network, receives input images of 416x416 and detects the angle orientation of the object.
4. Using the prediction from the previous step, rotate the code and send it to decoding framework.

The authors report a successful detection rate (after 8000 epochs and with a threshold of 0.5) of 0.958 and 0.888, for each of the used QR code datasets respectively.

Even though the used datasets are reported in the study, the authors do not provide the labels used and instead link to the papers where these were introduced. This papers, though, did not use *YOLO* and, as such, the label format is not compatible with it. More details on this can be found in chapter 5.

Chapter 3

Theoretical foundations: YOLO

YOLO (You Only Look Once) [28] is a CNN that, according to its authors, *"presents a new approach to object detection"*. This new approach can be explained in the fact that the authors propose to treat object detection as a *"a regression problem to spatially separated bounding boxes and associated class probabilities. A single neural network predicts bounding boxes and class probabilities directly from full images in one evaluation"*.

The main novelty that *YOLO* presented, thus, is the capability of performing detections in just one sweep (this feature gave *YOLO* its name: You Only Look Once). This simple architecture allowed *YOLO* to process 45 frames per second.

Several versions have been presented for *YOLO*. In section 3.2, the first version of *YOLO* is presented, and in the sections 3.3 and 3.4 the main changes introduced in the second and third version of *YOLO* are presented. Explaining the evolution of *YOLO* will allow to understand how it tackles the object detection problem, as well as explaining some of the decisions taken in this project and some of the future work that is proposed in section 9.

3.1 YOLO main characteristics

YOLO unifies all of the components of object detection into one single neural network. This allows it to predict bounding boxes using features from the entire image, and to predict all bounding boxes of all classes simultaneously.

YOLO divides the image in SxS different cells, which are responsible for the prediction of the objects that fall within each of them (i.e. if the center of an object falls within a certain cell, that section is the responsible of its prediction). Each of these cells predicts a total of B bounding boxes and its associated confidence scores. The confidence score is defined as the

$$Pr(Object) * IoU_{pred}^{truth}$$

. $Pr(Object)$ is the class probability, whereas $IoU_{pred}^{t}ruth$ is the Intersection over Union between the predicted bounding box and the ground truth.

The format for each bounding box is: x, y, w, h and *confidence*. The (x, y) coordinate represents the center of the bounding box, and all of the coordinates contain values in the interval [0, 1], as

their values are represented with resepect to the size of the image.

The representation of each bounding box and the SxS grid yields a tensor of predictions $S * S * (B * 5 + C)$, being C the number of classes that the model detects. Therefore, varying the number of classes also varies the output tensor size.

The loss function used in YOLO is the following:

$$\begin{aligned}\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x_i})^2 + (y_i + \hat{y_i})^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\ + \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S^2} 1_{ij}^{obj} \sum_{C \in classes} (p_i(c) - \hat{p}_i(c))^2\end{aligned} \tag{3.1}$$

Each of the terms can be interpreted to understand the reason why the authors chose this loss function.

The first term, which is:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x_i})^2 + (y_i + \hat{y_i})^2] \tag{3.2}$$

is the Sum of Squares Error (SSE) between the predicted bounding box coordinates and the ground truth coordinates. The term 1_{ij}^{obj} is equal to 1 if an object appears in the cell i and the cell is responsible of its prediction (i.e. has the higher IOU). The authors use the SSE because, in their own words, *"it is easy to optimize, however it does not perfectly align with our goal of maximizing average precision"* [28]. The term λ_{coord}, which is equal to 5, is used to increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that do not contain objects. This can prevent model instability problems.

The second term, which is:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{3.3}$$

is similar to the first one, and is used to calculate the error in the predicted bounding box dimensions. The square root is used to reflect that small deviations in large boxes matter less than in small boxes.

The third term, which is:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 \tag{3.4}$$

represents the confidence error, where $0 \leq C_i \leq 1$, and $\hat{C}_i = 1$.

The fourth term, which is:

$$\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 \tag{3.5}$$

is used when there is no object in the grid. In this case, the classification and localization errors need not to be taken care of, and the only parameter to take into account is the confidence C. For that, the variable 1_{ij}^{noobj} is used. It equals to 1 when there is no object in the cell, and 0 otherwise. This term pushes the confidence scores of the cells with no objects towards zero.

The fifth and last term is:

$$sum_{i=0}^{S^2} 1_{ij}^{obj} \sum_{C \in classes} (p_i(c) - \hat{p_i}(c))^2 \tag{3.6}$$

which sums the errors of all the classes probabilities for each of the grid cells.

3.2 YOLO design

YOLOv1 has a total of 24 convolutional layers (which extract features), followed by 2 fully connected layers (which make the predictions based on the extracted features. The image input is 448x448 pixels.

Each of the cells in the grid makes a prediction, and therefore the number of predictions is limited to 98, because each of the 7x7 cells predicts 2 bounding boxes (which causes close object detection issues).

Non-max suppression is used to remove repeated repetitions (an object may be detected by different cells at the same time, and therefore, the repeated bounding boxes should be pruned so that only the "best" one should remain).

Some of the limitations of this first version of *YOLO* are the struggle to localize objects correctly (as shown in the experiments performed in the VOC 2007 dataset in [28]) and the strong spatial constraint on bounding box predictions (due to the previous explained fact that only 2 predictions are made per cell).

3.3 YOLOv2

On December 25th of the same year, the original authors published *YOLOv2* [29], which introduced some changes to make *YOLO* more accurate and faster.

The main introduced changes are:

1. **Batch Normalization**: Adding this allowed an improvement in convergence and improved the mAP in more than 2%.
2. **High Resolution Classifier**: In *YOLOv1*, training was performed with an image input of 224x224, and then inference was performed with an image input of 448x448. In this new version, the classification network is first tuned at an image input size of 448x448 during 10 epochs in *ImageNet* [30].
3. **Anchor Boxes**: In *YOLOv1*, the coordinates of bounding boxes were predicted with a fully connected network on top of the feature extractor. In *YOLOv2*, this is changed and anchor boxes are used instead.

4. **Reduced input size**: The input size is reduced to 416x416 pixels, which yields a feature map of 13x13. This input is not fixed; it is changed every 10 batches to force the network to learn how to predict well across a variety of input dimensions.

5. **Increased bounding boxes predictions**: By using anchor boxes, *YOLOv2* is able to increase the bounding box prediction count from 98 to more than a thousand.

6. **Darknet 19**: A new architecture with 19 convolutional layers is used.

3.4 YOLOv3

On April 8th, 2018, *YOLOv3* the original authors published the paper with what would be their last update[1]: *YOLOv3* [31].

The main introduced changes were:

1. **Objectness score predicted using logistic regression**: The objectness score should be 1 if the bounding box prior overlaps a ground truth object by more than any other bounding box prior (i.e. its the best candidate to make the prediction).

2. **Multilabel precition**: Each bounding box now uses independent logistic classifiers instead of softmax to make class predictions, which helps when making predictions in datasets with overlapping labels (i.e. Dog and German Sheperd).

3. **Predictions across scales**: *YOLOv3* makes predictions at 3 different scales, which yields a resulting tensor $NxNx[3*(4+1+80)]$ for the 4 predicted bounding box offsets, the objectness prediction and the 80 class predictions (in the case of the *COCO* dataset).

4. **Darknet 53**: A new architecture with 53 convolutional layers is used (as shown in figure 3.1). It uses successive 3x3 and 1x1 convolutional layers with shortcut connections [32].

From this moment, for simplicity *YOLOv3* will be referred to as *YOLO*.

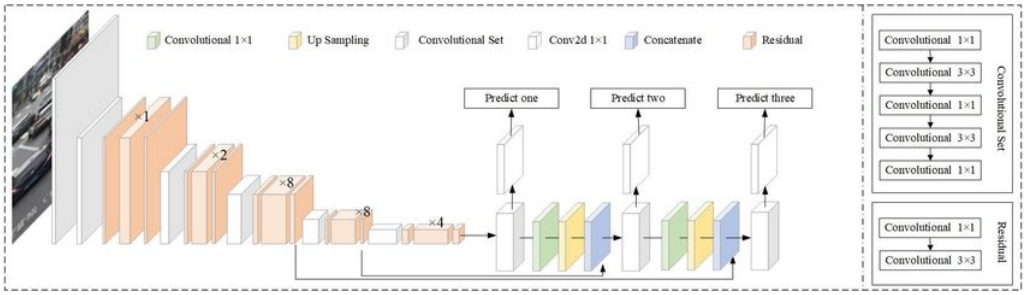


Figure 3.1: *YOLOv3* architecture. (Source: [33])

[1] Joseph Redmon, one of the creators of *YOLO*, retired from the field of Computer Vision citing ethical concerns. Food for thought: https://twitter.com/pjreddie/status/1230524770350817280

Chapter 4

Proposed solution

The most common methods for detecting QR Codes are based on classic computer vision methods. Recently, though, an interest has been put on studying how Convolutional Neural Networks perform in this task [17] [16] [34].

One of the main problems with the classic computer vision approach is that deformations in the QR Code can make it impossible to detect or decode it. In fact, this is the main issue that motivated this project.

Now that the main CNN architectures have been seen in section 2, the proposed solution is to use *YOLO* as baseline to detect QR codes. *YOLO* is a general detector which seems to have the better combination of accuracy and speed among all the architectures presented in 2, and the idea of applying it to detect QR codes is not new. Nonetheless, very few papers researching this idea are available, such as [27] (see section 2.2.6), and these do not provide much detail. Furthermore, there are no papers (to the knowledge of the author) that study how the most recent version of *YOLO* (see section 3.4) performs in this task.

After seeing the previous work done in QR code detection with *YOLO*, it is clear that much work must be done to study this, which will require a lot of time. Due to the data preprocessing issues that were suffered shortly after the project began (explained in chapter 5) it has been considered that only QR codes detection would be studied in this project, and that QR codes segmentation would be left as future work (see section 9.3 for more details).

YOLO has been chosen over *SSD* because of the latest versions (including the fourth one, released on April 23rd and which is discussed in section 9.6) improve their performance by a nice margin. Furthermore, *YOLO* may be used in combination with *YOLACT* in the future, which is a nice possible feature that could come in handy (see section 8.8 for full details on this).

Three main novelties are introduced by this project:

1. Several QR Code datasets labeled with a *YOLO* compatible format. Although some papers have used *YOLO* with QR Codes, the datasets used are not publicly available to the knowledge of the author.

2. A study of how *YOLO* performs in the task of detecting QR Codes. There are already some publications regarding this, like [27]. This project, though, aims to give a deeper study on how

YOLO performs, including an analysis of how image augmentations can help in the process.

3. A study of how *YOLO* can handle the classification of QR versions. A paper studying this is not publicly available to the knowledge of the author.

In chapter 3, the theory behind *YOLO* has been explained. Chapter 5 explains the datasets used in the experiments, whereas chapter 6 details the most interesting parts of the implementation process, so that the reader can have a grasp of the issues that he or she may encounter when replicating this work.

Chapter 5

Data preprocessing

To the knowledge of the author, no QR dataset containing YOLO labels are publicly available in the internet.

This represents one of the main issues in data science projects: the absence of publicly available datasets. This has been discussed in papers such as [35], and is a problem that has been suffered in this project as well. This, in turn, has reduced the amount of time spent in actually processing these datasets to get some valuable results, and is a problem that should be tackled to help in the progress of the data science field.
Even though there is a limitation that is not easy to overcome, as many institutions still see open science and tools as a detriment to themselves, data scientists still have to try to change this situation. I firmly believe it benefits no one and, more importantly, it slows progress down (think in the immense amount of time that has been spent in generating the same kind of datasets over and over again simply because of the fact that the people that had them did not release them).

In [27], the authors describe the use of *YOLO* to detect barcodes and QR codes and specify two datasets that were used in the process, but upon downloading one of the two (since the other is not available) it was found that the label format is incompatible with *YOLO*, as no bounding box is present. Therefore, it was clear that the creation of one or more labeled datasets is necessary to proceed with this project.

Two different sets of labels will be generated for each image: in the first one, the QR version will not be stored as a class, and in the second one it will. The idea behind this, as explained before, is to train the model with the same set of images with QR version as a class and without it to study how well it is able to perform QR codes detection and, separately, classify each version.

A dataset with "real" images obtained in a Universitat de Barcelona event [36] was expected to perform some experiments in a real scenario but unluckily due to the COVID-19 situation the event was cancelled, which this forced that the experiments were only performed in synthetic datasets or with images obtained in the internet. This increased the amount of time spent in the generation of the datasets, as they would be the only ones used and not just replacements until the "real" dataset could be available.

After generating these two datasets, a third one was created to specifically study the performance

of *YOLO* in classifying QR code versions. This is explained in further detail in section 5.4.

5.1 YOLO label format

YOLO's label format is the following: $\{class\ x\ y\ w\ h\}$, being *class* the class identifier of the object present in the bounding box, x and y being the center coordinates of the object, and being w and h the width and height of the bounding box, respectively.

The coordinates are normalized with respect to the size of the image, and therefore all of their values are contained in the [0...1] interval.

The format used when generating the datasets of sections 5.2, 5.3 and 5.4 is the same as *YOLO's*.

5.2 Dataset 1

In a first iteration, two datasets obtained from [37] were used. In total, the two contain 810 images. A script was written to automatize the process, which has the following steps:

1. Resize the image to 416x416 pixels. The resultant image keeps the aspect ratio, and this is ensured by using black background to fill the empty spaces. This is done because the chosen implementation (see section 6.2) needs images with this size to train.
2. Using an internal *ColorSensing* module, try to find and decode a QR present in the image.
3. If no QR has been found, discard the resized image. If a QR has been found, store the resized image and the two label files, all of them with the same name in their respective folders (i.e. "images", "labels" and "labels no class".
4. The probability with which an image is assigned to the training, validation or testing dataset is 60%, 20% and 20% respectively. Numpy's random choice [38] has been chosen for this process.

This resulted in a total of 270 labeled images. As the objective was to have at least about 300 images in total, 55 extra synthetic images containing codes of versions 1 to 4 were added by processing them with the previously defined script. All the synthetic images were successfully decoded, resulting in a total of 325 images, divided in 193 images in the training set, 62 images in the validation set and 70 in the testing set.

For the 55 synthetic added images, QR codes with classes 1 to 4 were chosen because these versions were lacking in the dataset, and since are very common, having a similar amount of examples of these compared to the bigger versions was considered important.

The distribution of QR versions in this dataset is the following:

QR Version	Training	Validation	Testing
1	7	3	3
2	11	2	2
3	15	3	5
4	29	6	19
5	37	18	13
6	40	15	13
7	27	7	6
8	24	2	4
9	2	4	3
11	0	2	0
15	1	0	1
16	0	0	1

For simplicity, from now on this dataset will be referred to as *Dataset 1*.

The QR codes with versions 11, 15 and 16 were preserved because of the small amount of data available. Even though there are not enough to split into the three datasets, it seems it may be preferable to keep them to allow the feature extractor to have some extra samples, even though these will probably not be usable to learn to identify these versions. Furthermore, for the experiments in which no classes are used these samples will not pose any problem.

5.3 Dataset 2

In a second iteration, all of the images of a dataset containing 10.000 synthetic images [39] were added to the first batch of images. Almost all of these were successfully decoded, which resulted in a dataset containing a total of 10270 images, divided in 6160 images in the training set, 2085 images in the validation set and 2025 images in the testing set.

The distribution of QR versions in this dataset is the following:

QR Version	Training	Validation	Testing
1	1487	517	496
2	1499	486	516
3	1522	490	497
4	1524	542	474
5	41	15	12
6	43	13	12
7	18	13	9
8	19	4	7
9	4	5	0
11	1	0	1
15	1	0	1
16	1	0	0

For simplicity, from now on this dataset will be referred to as *Dataset 2*.

5.4 Dataset 3

The experiment done in section 7.6 using the *Dataset 2* showed that a dataset with a more balanced class distribution was needed. In order to achieve it, a third dataset was generated, which will be referred to as *Dataset 3*.
To create this process, a script was written. The main steps of the script were the following:

1. For every QR code version from 1 to 13 (both included), iterate 950 times.
2. Select a random correction value out of the 4 possible ones (see section 1.2.6).
3. Generate enough random data to fill the specific QR code version being process.
4. Create a QR code with the aforementioned parameters and place it in the center of a 416x416 white image.
5. Generate the appropriate *YOLO* compatible label.
6. Store both the image and the label in split files with the same names (i.e. *1.txt* for the label and *1.png* for the image).

This dataset contains a total of 9750 images with QR Code versions ranging from 1 to 13 (both included). There are a total of 750 codes of each version, which yields the resulting 9750 images.

As explained before, *Dataset 3* has been generated with completely synthetic images that contain a QR code that codifies random data and uses a random correction method (the combination of the two guarantees that the variable part of each code will be different). This guarantees that the QR codes belonging to the same version have some features in common (as the common elements in a specific QR code version are the same, as explained in section 1.2) but that they are not exactly the same. This maximizes the amount of different samples while allowing some common features that can be used to correctly identify each version.

The QR code is located at the center of the image. Even though these "perfect" conditions may never happen in a real scenario, it is accepted since the main use of this dataset will be to study whether if *YOLO* is able to correctly identify each of the versions of QR codes. Therefore, it is interesting to begin with the easiest scenario, since if it is not able to perform that task with these conditions, it will not be able to do so in tougher scenarios.

5.5 Datasets availability

The three datasets used are publicly available at https://www.kaggle.com/hamidl/yoloqrlabeled.

5.6 Image augmentation

Data augmentation tries to reduce the problem of overfitting by tackling the root of the problem: the training dataset. The augmentations artificially increase the size of the training dataset by either data warping (a transformation in which the label is preserved) or oversampling (create synthetic instances and add them to the training dataset). For this project data warping is used, since the original images are augmented.

The library imgaug [40] has been chosen to generate the image augmentations. imgaug is a Python library that *"converts a set of input images into a new, much larger set of slightly altered images"*.

Initially, torchvision [41] was chosen as the library with which to perform augmentations but it was later discarded because even though its integration with PyTorch [42] made it ideal for this use, it does not adapt the labels to the augmented image, a necessary feature to use it in this project.

The features that made imgaug the chosen library was that it transforms the label as well as the image to generate the augmented image, as well as the vast types of augmentations (artistic, blur, blend, color, etc.) and the fact that it worked with bounding boxes and segmentation maps.

Since *Dataset 1* and *Dataset 2* contain real images and were used in experiments with different objectives in mind as *Dataset 3*, two image augmentation pipelines have been created.

The first one tries to imitate images that are expected to be found in a production environment in which both QR codes and their versions want to be detected. Therefore, its main task will be to apply rotations and perspective transformations.

The second image augmentation pipeline has been created with the main task of varying the sizes of the QR codes present in *Dataset 3* to prevent the QR codes of the same version to have the exact same size and thus be used in its classification. Therefore, its main task will be to apply resizing transformations.

5.6.1 First image augmentation pipeline

The expected images that a production system would receive would most probably have perspective and cylindrical deformations, as the QR codes will be attached to either bottles (which would give them the cylindrical deformation) or to flat surfaces out of which the user would take a picture, not necessary with an orthogonal angle with respect to the plane of the QR (and, therefore, it would probably contain a perspective deformation). Knowing this, the image augmentation algorithm should try to generate these kind of deformations, as they would make the distribution of the images as similar as possible to those of the production environment.

Specifically, the image augmentation pipeline is the following:

1. **GammaContrast**: Adjusts image contrast by scaling pixel values to $255 * ((v/255) ** gamma)$.
2. **Fliplr**: Flips 20% of the images horizontally.
3. **Rotate**: Applies affine rotation on the y-axis to input data.
4. **PerspectiveTransform**: Applies four point perspective transformations to images.
5. **Affine**: Scales images to a value between 80% and 120% of their original size, translates images by -10% and 10% on the x-axis and y-axis and fills new pixels (if any) with a constant value.

As there is no "direct" method to apply cylindrical deformations with imgaug, only perspective ones have been taken into account for the experiments.

The augmentations have been applied online: the image is augmented every time it is read. Some examples of an original image and its generated augmentations can be seen in figure 5.1.

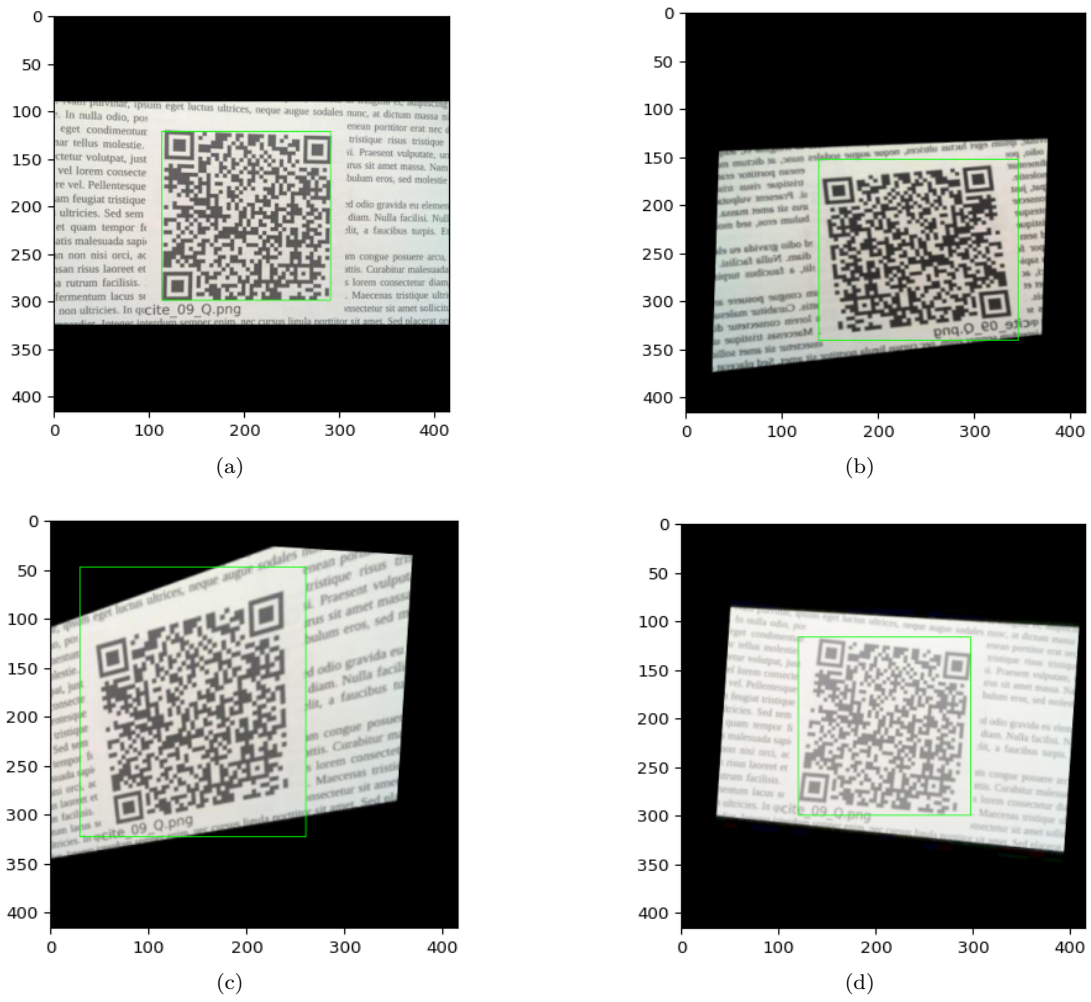


Figure 5.1: (a) Original image (b) Augmentation produced with the first image augmentation pipeline (c) Idem (d) Idem

5.6.2 Second image augmentation pipeline

As explained in section 5.4, the generated QR codes in *Dataset 3* are centered and, as section 1.2.1 explains, the QR codes of different versions have different sizes. Therefore, each version of a QR code will have a different size, which could be used by *YOLO* in the classification process. To prevent this, the following image augmentation pipeline has been defined:

1. **ScaleX**: Scales images along the width to sizes between 80% and 120%.
2. **ScaleY**: Scales images along the height to sizes between 80% and 120%.

The random value by which both axes are scaled is the same to preserve the ratio.

The augmentations have been applied online: the image is augmented every time it is read. Some examples of an original image and its generated augmentations can be seen in figure 5.2.

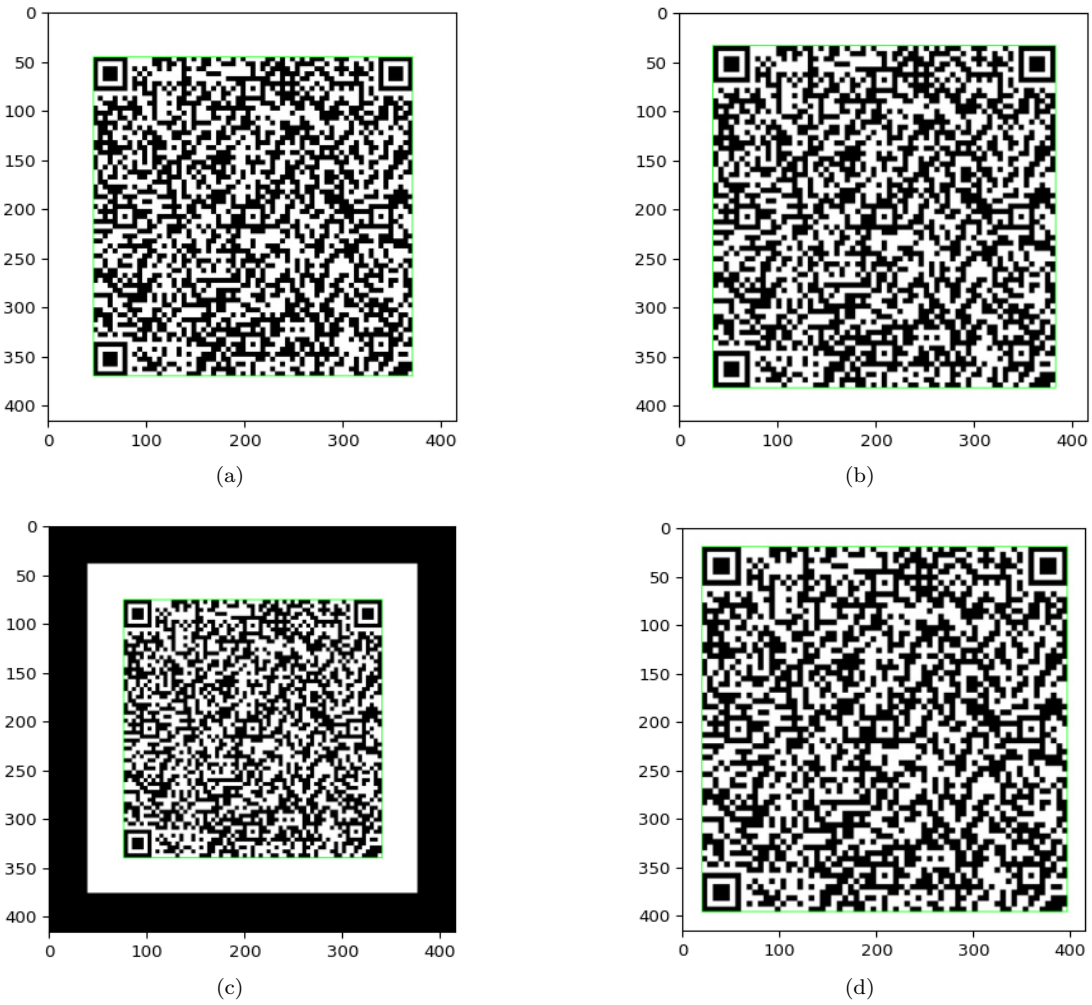


Figure 5.2: (a) Original image (QR code version 12) (b) Augmentation produced with the second image augmentation pipeline (c) Idem (d) Idem

Chapter 6

Implementation

This section describes the implementation process of the main module, as well as all the extra necessary components (for example, the logging functionality).

6.1 Requirements

To perform the objectives described in section 1.5, a *YOLO* must be used. It has been chosen to use an already implemented solution to spare the time in its implementation and invest it in modifying an already existing one to do what is needed for this project.

A simple implementation would allow an easy modification. By "simple", it must be understand that it does not have any extra features that would make the design excessively complex (i.e. it has the feature of performing detections with video) since these would not be used for this project, which is more focused in having a prototype with which research how it performs rather than delivering an immediate viable product.

Image augmentations were required. These have been explained in detail in section 5, so they will not be discussed here.

Producing the outputs of the images with the predictions was also needed, since these would be used in the discussion of the experiments, as well as storing the weights in a file to be able to use them later on in case it was needed.

The implementation should also have enough performance metrics already implemented, since these were also necessary in the discussion of the experiments to be able to do quantitative analyses.

Another requirement was logging, since an evaluation of performance was required. In particular, the comparison between the training and validation loss values was especially interesting to be able to understand how the network was training and when would the overfitting begin.

6.2 Tools used

Therefore, it was decided that the implementation that would be use would be one implemented in PyTorch. Specifically, [43] was chosen. The main reason to choose this over other popular implementations was that this one was simple enough to be studied and modified as needed, which fit the requirements explained in section 6.1.

To produce the logs, TensorBoard [44] was considered the ideal tool due to its integration with PyTorch. The chosen implementation had methods already available to produce TensorBoard logs, but they contained two main issues: 1) they used TensorFlow, and as such both PyTorch and TensorFlow needed to be installed and 2) some errors arose during the execution.

6.3 Discussion

Initially, the "original" implementation of *YOLO* was explored. Since it is the author implementation, it seemed a good idea to use it. Furthermore, it is implemented in C and CUDA, which means that it will perform better than the available implementations with either TensorFlow [45] or PyTorch [42], as these libraries are just Python wrappers to the actual implementations, which are written in C++ (and this makes them perform worse[2]). This implementation, though, proved difficult to use, especially because modifications of C and CUDA code are complicated. Therefore, a solution implemented with PyTorch or TensorFlow was deemed necessary. Furthermore, these frameworks provide extra tools (TensorBoard for logging, for example) that would be needed afterwards.

Since the repository has not received any updates since May 2019, some fixes were necessary to solve some of the issues that arose when using it. Among these errors, there were many issues with training, as the Python process responsible of the Pytorch execution was killed several times due to memory issues by Linux. When analyzed, the reason behind this issue was found: the process was using much more memory than the system had, which was quite strange (considering that the system has 32 GB available, as explained in section 7.2).

This error was solved by using Python's garbage collector, which was used to free non-used memory. This solution fixed the problem and saved many hours of training (and considering that this error sometimes happened after several hours of training this is quite noticeable).

Some other minor errors were also found, but since are not so interesting as the previously explained, they will be omitted from this section.

To produce the logs, TensorBoard [44] was considered the ideal tool due to its integration with PyTorch. The chosen implementation had methods already available to produce TensorBoard logs, but these had two main issues: 1) they used TensorFlow, and as such both PyTorch and TensorFlow needed to be installed and 2) some errors arose during the execution.

PyTorch requires about 1.4GB of disk space, whereas TensorFlow requires about 1.5GB, so having TensorFlow installed just for the logging was nonsense.

[2] A good explanation on the issues with Python for Deep Learning in this [46] FastAI post

As such, the logging functions needed to be improved. The decisions made were: clear the dependency on TensorFlow to be able to use TensorBoard without the need of having the first installed (since PyTorch now includes TensorBoard functionalities built in, this was not too problematic) and create a new function exclusively to generate a visualization with the comparison between the training and validation losses. The validation loss function was not being calculated by the default implementation, so adding it was needed.

Chapter 7

Experiments

This section introduces the experiments and results obtained with several different datasets, as well as their results and an explanation of why they were made.

For all the experiments, the weights when the model had a highest validation mAP are stored. These weights are later used to compute the mAP in the testing dataset, as well as to make the inferences shown.

Several hypotheses are going to be studied in this section:

1. **Can transfer learning be used to perform QR codes detection?** This will be studied in section 7.3. In these experiments, the performance of transfer learning will be studied, and a decision will be made: to use it or not for the rest of the experiments. It would be interesting to perform all the experiments both with and without transfer learning, but due to time constraints, only one setup can be studied. Therefore, it is necessary to do this before any of the successive experiments can be performed.

2. **Can *YOLO* detect QR codes?** This will be studied in sections 7.4 and 7.5. It has been split in two sections because this task is first performed without image augmentations. Afterwards, image augmentations are applied to check whether them improve or not the obtained results.

3. **Can image augmentations be used in combination of synthetic images?** As explained in section 5, synthetic images comprise the core of *Dataset 2*. This arises the hypothesis of whether if image augmentations can be used to try to mitigate this issue by trying to generate images as close as those expected to be found in "real" scenarios.

4. **Can *YOLO* classify QR codes by their version?** This will be studied in sections 7.6 and 7.7. In these experiments, it will be studied whether *YOLO* is able of correctly classify the versions of QR codes (see section 1.2.1) or not. For these experiments, fully synthetic datasets will be used with one idea in mind: if *YOLO* is not able to detect them in images with "perfect conditions", it will not be able to do so in a production environment.

5. **Can a shallower and faster version of *YOLO* be used to classify QR codes by their version without loosing too much performance?** This will be studied in section 7.9. In this experiment, *YOLO-tiny* will be used. This shallower version of *YOLO* is able to train and perform detections much faster, which would be a very interesting property, but at the cost

of a loss in performance. This experiment will try to check whether if *YOLO-tiny* is able to correctly classify the versions of QR codes. For these experiments, fully synthetic images will be used as well with the same idea in mind as previously explained.

The aforementioned hypotheses can be used to determine whether the objectives explained in section 1.5 can be achieved or not. Due to time constraints, it has been decided that the segmentation of QR codes will not be studied to be able to study in more depth the detection of QR codes (i.e. how good the is at detecting QR codes, how good is it at detecting QR code versions and how fast it can get without loosing performance).

7.1 Performance metric

To study how the results of the following experiments perform and decide which one is better, a performance metric needs to be chosen. Since the *mean Average Precision (mAP)* is a very common performance metric in Computer Vision, the experiments in the subsequent sections will use it.

To define it, the Intersection over Union must be described first. The IoU is the proportion given by the ratio of the area of the intersection and area of union of the predicted bounding box. Figure 7.1 gives a visual example.

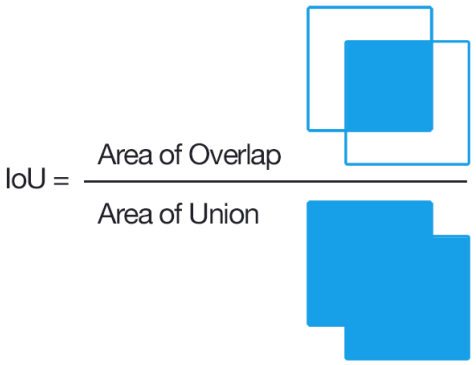


Figure 7.1: Intersection over Union. (Source: [47])

The IoU is used to determine if the prediction is a:

1. **True Positive**: The IoU is greater than the threshold value and the predicted class matches the ground truth.
2. **False Positive**: The IoU is lesser than the threshold value.
3. **False Negative**: The IoU is greater than the threshold value but the predicted class in incorrect, or no prediction has been performed where a ground truth was present.

A typical IoU value in computer vision is 0.5 (as the one used in this project; see section 7.2).

The *Average Precision* (AP) is obtained by calculating the area under the precision-recall curve, which describes how the *precision* (the number of correct guesses by the model) and the *recall* (if the model has guessed correctly the number of times it should have guessed) vary.

7.2 Experimental setup

All the experiments have been performed in a personal computer with a Nvidia RTX 2080 SUPER GPU, which has 8GB of VRAM, an AMD 3900X processor, which has 12 cores and 24 threads and 32GB of RAM memory (due to the fact that because of the COVID-19, a server bought to perform the experiments but it could not be installed before the company facilities were closed).

The IoU threshold has been set to 0.5 for all the experiments. The confidence threshold has been set to 0.5 (i.e. the detections with a confidence interval lower than 0.5 are removed) and the NMS (Non Maximum Supression) threshold has been set to 0.5 for the NMS process.

The learning rate has been set to 0.001, and remains fixed during the entire training process. The decay is set to 0.0005 and the momentum is set to 0.9.

Adam [48] has been used as the optimizer.

In sections 7.3, 7.4, 7.5 and 7.6, the testing dataset from *Dataset 1* will be used as testing dataset, both when training with *Dataset 1* as when training with *Dataset 2*.

Furthermore, in the experiments in which *Dataset 1* is used more epochs will be run in comparison with the experiments in which *Dataset 2* is used. The reason behind this is that the first dataset contains way less images (as explained in 5), which means that each epoch takes less time to run and, therefore, more epochs can be performed in the same amount of time. Since the experiments have been run in a personal computer (as explained in section 7.2), unfortunately, the training time has been less than what would have been desired.

In sections 7.7, 7.8 and 7.9, the testing dataset from *Dataset 3* will be used as testing dataset.

7.3 Transfer learning

7.3.1 Settings

As stated in [49], *"the study of transfer learning is motivated by the fact that people can intelligently apply knowledge learned previously to solve new problems faster or with better solutions."*

Therefore, it is interesting to check whether a feature extractor trained with a common dataset (like *ImageNet*) can be used to detect QR codes. To study this, the weights of *Darknet 53* (*YOLO*'s backbone) pretrained with *ImageNet* can be used to obtain a good performance.

The *ImageNet* dataset contains more than 14 million images (more specifically, a total of 14.197.122 images [30]) and more than 20 thousand categories. QR codes are not contained as a class in it, and as such we believe that training a custom feature extractor with the datasets explained in section 5 will be better than using a feature extractor trained with *ImageNet*.

To test that, pretrained *Darknet 53* have been downloaded from [50] and used to train the classifier. Two tests will be performed, one in which the weights of the backbone will be freezed (figures 7.2 and 7.3) and one in which the weights will not be freezed (figures 7.8 and 7.9).

The classifier weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

For all these experiments, *Dataset 1* and *Dataset 2* are going to be used. For testing, the testing dataset of *Dataset 1* is going to be used.

No image augmentations were applied, either when freezing the pretrained backbone as well as when the pretrained backbone was used to initialize weights.
To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. When using the *Dataset 1*, a total of 1000 epochs were run, whereas when using the *Dataset 2*, a total of 120 epochs were run.

As explained in section 7.2, the difference between the epochs is due to the available hardware.

For readers without much experience in machine learning, [51] may be useful to understand the graphs shown in the results section, both for this experiment as for the performed in the following sections.

7.3.2 Results: freezed backbone

When freezing the *Darknet 53* backbone, the obtained training and validation losses are the following:

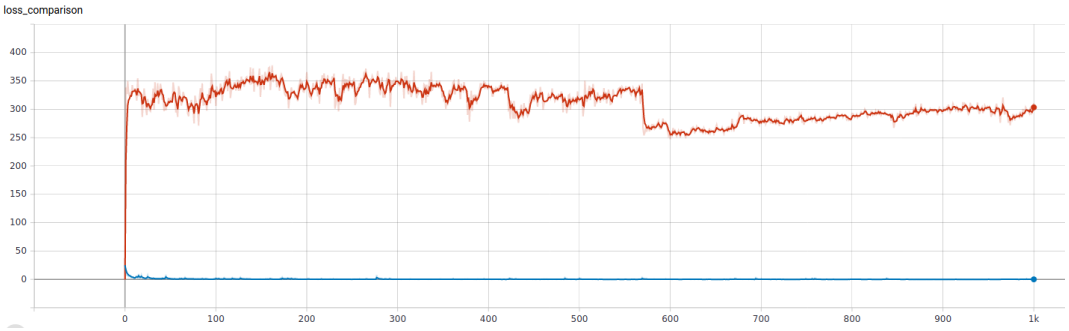


Figure 7.2: Training and validation loss functions when training with *Dataset 1* and using the freezed pretrained *Darknet 53* backbone with *ImageNet*.

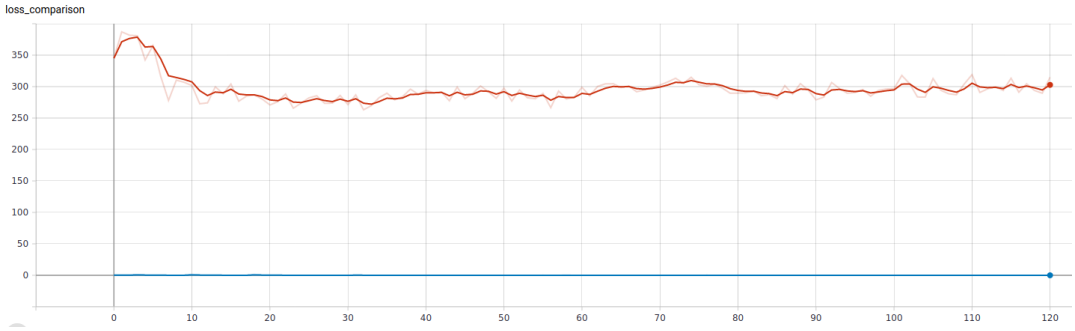


Figure 7.3: Training and validation loss functions when training with *Dataset 2* and using the freezed pretrained *Darknet 53* backbone with *ImageNet*.

And the obtained mAPs are:

Metric	**Dataset 1**	**Dataset 2**
mAP	0.967	0.985

Table 7.1: Model performance metrics after training with both datasets and using the freezed pretrained *Darknet 53* backbone with *ImageNet*.

7.3.3 Discussion: freezed backbone

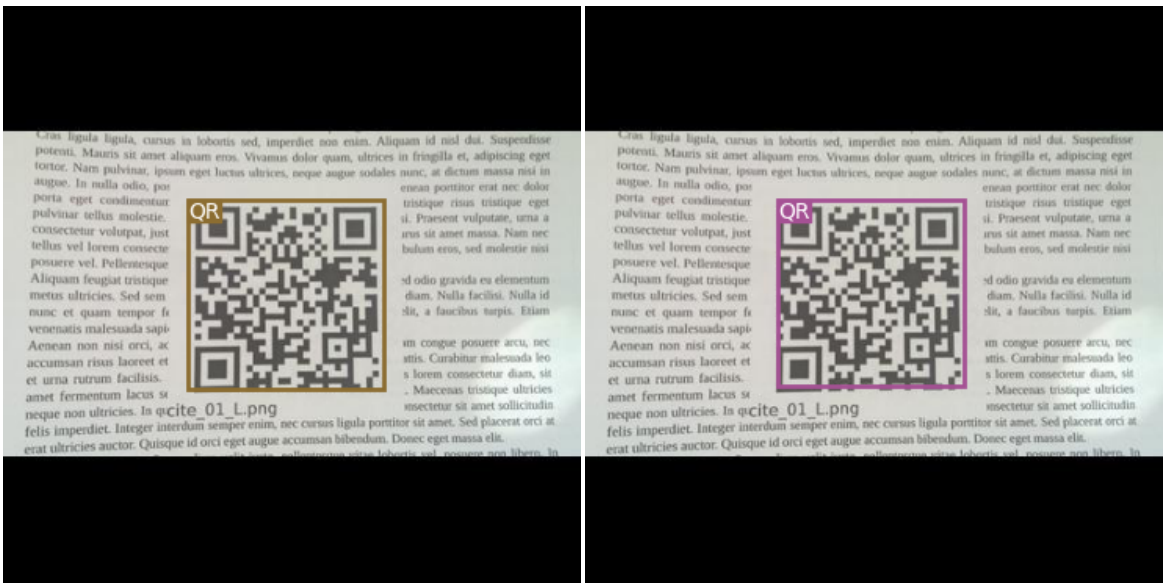
The validation losses are higher when using the pretrained *Darknet 53* backbone, as can be seen when comparing figures 7.2, 7.3 and figures 7.10, 7.11.

These results hints that the model trains better when not using the pretrained *Darknet 53* backbone, as the loss function takes into account the sum of squares error between the predicted bounding boxes and the ground truth coordinates, among other things (as seen in section 3.1). Therefore, it must be assumed that the higher validation losses are caused by bounding boxes predictions which are farther to the "real" value.

Furthermore, when training the backbone from scratch, the obtained validation losses seem to be more stable, which hint that training the backbone from scratch provides better results.

Moving on to the obtained mAP, the results obtained in table 7.1 seem to be better than those seen in table 7.3. Even though the mAP obtained in *Dataset 1* are almost equal, when comparing them in *Dataset 2* the results obtained when using a pretrained backbone are better. The reason behind this is not very clear, but it may most probably be due to the fact that the features extracted from synthetic images are not of much value when making predictions in a testing dataset containing real images.

For the reasons explained in the *Rebuttal* section of [31], it is interesting to see visually the results rather than just using the mAP. Therefore, let's take a look at some inferences performed by both models:



(a) (b)

Figure 7.4: (a) Detection by model trained with *Dataset 1* and transfer learning (b) Detection by model trained with *Dataset 2* and transfer learning

(a) (b)

Figure 7.5: (a) Detection by model trained with *Dataset 1* and transfer learning (b) Detection by model trained with *Dataset 2* and transfer learning

(a) (b)

Figure 7.6: (a) Detection by model trained with *Dataset 1* and transfer learning (b) Detection by model trained with *Dataset 2* and transfer learning

It seems that some of the bounding boxes are more accurate when training the feature extractor and using the *Dataset 1* (figures 7.12, 7.13 and 7.14) are better than those produced when using transfer learning (figures 7.4, 7.5, 7.6), as the bounding boxes are more adjusted to the QR codes. When training with the *Dataset 2*, transfer learning proves its value and the inferences are better, since no multiple predictions are made. There are some predictions, though, where the predicted bounding boxes are larger than the QR code, which hints that using transfer learning also has some problems (figure 7.7).

Figure 7.7: Inference produced when using transfer learning and the *Dataset 2* where the predicted bounding box is greater than the QR code.

7.3.4 Results: non-freezed backbone

When not freezing the backbone, the obtained training and validation losses are the following:

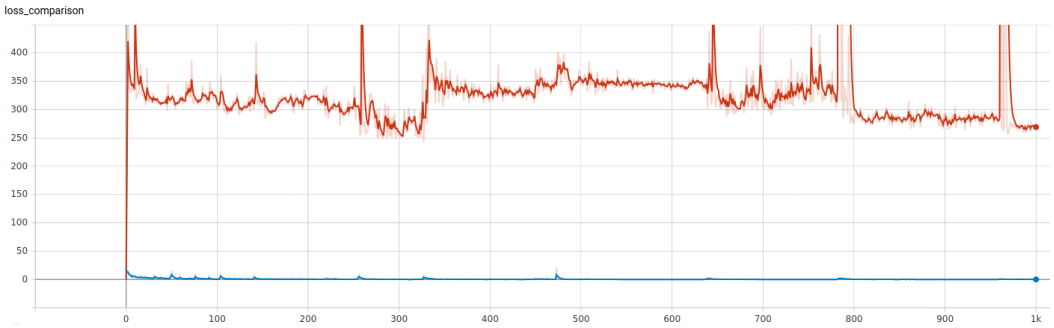


Figure 7.8: Training and validation loss functions when training with *Dataset 1* and using the non-freezed pretrained *Darknet 53* backbone with *ImageNet*.

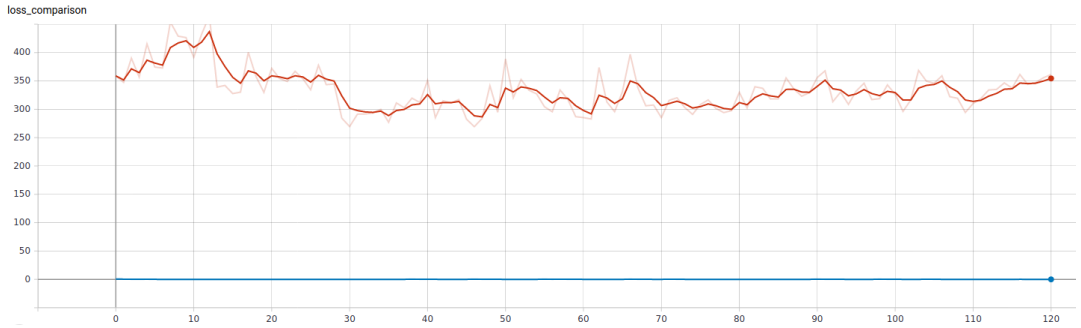


Figure 7.9: Training and validation loss functions when training with *Dataset 2* and using the non-freezed pretrained *Darknet 53* backbone with *ImageNet*.

The obtained mAPs are:

Metric	Dataset 1	Dataset 2
mAP	0.956	0.949

Table 7.2: Model performance metrics after training with both datasets and using the non-freezed pretrained *Darknet 53* backbone with *ImageNet*.

7.3.5 Discussion: non-freezed backbone

The validation loss is way more unstable when training with *Dataset 1* using the non-freezed pretrained *Darknet 53* backbone, and its value is also higher until the last 200 epochs (see figures 7.8 and 7.10). When using *Dataset 2*, the validation loss is also more unstable, with a constant increase until the 12th epoch and a decreasing from that point onward (see figures 7.9 and 7.11).

The validation and training losses seem to hint that not freezing the pretrained *Darknet 53* backbone offers a worse performance. The reason behind the observed spikes in figure 7.8 are unknown. They hint that, for some reason, there are extreme deviations between the predictions and the ground truths, which is fixed short after. Since these extreme spikes are only observed in this section, it seems that they may be related to the fact that the unfreezed pretrained backbone is being used. Trying different hyper-parameters and a different optimizer may reduce them. This should be further studied to understand the reason behind it.

Comparing the obtained mAPs, the results seen in table 7.2 show that the results are quite good. It is interesting to see that the mAP slightly decreases when training with *Dataset 2*. The reason behind this may be due to the fact that the testing dataset of *Dataset 1* is being used for both experiments, and since almost no synthetic images are present there, the extra synthetic images added to *Dataset 2* in comparison to *Dataset 1* do not provide many useful features for the predictions.

When comparing the results obtained in table 7.2 to the results seen in table 7.1, it is seen that the performance is slightly worse both training with *Dataset 1* and when training with *Dataset 2*. This may be due to the fact more training is needed (take into account that training was performed during 120 epochs) or that the synthetic images added in *Dataset 2* do not provide much value.

7.3.6 Discussion: transfer learning

At a first sight, it seems that it may be interesting to use transfer learning when small datasets are available to train the feature extractor but not with large datasets, as it may be the case with datasets as big as *Dataset 2* and *Dataset 3* (sections 7.7 and 7.8) and, especially, with even larger ones.

Nonetheless, the results obtained when using *Dataset 1* proves the contrary. When using that dataset, the results obtained with a backbone trained from scratch seem to be better than when using transfer learning.

On the other hand, when using a bigger dataset such as *Dataset 2* transfer learning provides better results than training the backbone from scratch which, again, is not what was expected.

One extra issue thay may worsen the results obtained when training the backbone from scratch is training time. It is not stated for how many epochs the pretrained *Darknet 53* has been trained, but taking into account previous *Darknet* backbones versions trained by the same author, it may be assumed that it has been trained for 160 epochs in *ImageNet* (as with *Darknet 19*; see [28]). This is much more than the performed in sections 7.3.2 and 7.3.4. Therefore, it may be assumed that with more training (which could not be done because of the hardware available), the results could improve.

Furthermore, this project does not aims to provide a functioning prototype, but is more oriented to be a starting point in the research of how *YOLO* performs when detecting and classifying QR codes. As such, training the custom backbone from scratch seems more interesting than using transfer learning.

In this project, and due to the aforementioned reasons and issues, transfer learning will not be used, and the feature extractor will be trained with the QR codes datasets instead.

Nonetheless, it must be noted that even though it has not been done in this project, it would be interesting to perform all of the experiments in section 7 with transfer learning to see if there may be any scenario in which using it could be helpful.

7.4 QR codes without augmentations

7.4.1 Settings

After this, the next question that was necessary to answer was *"Can YOLO detect QR codes in images?"*. In [28] and [29], the authors study the performance of YOLO and show how well it performs in detecting and classifying objects. QR Codes, though, are not a class in any of the datasets that they train use in their benchmarks, so even though it seems that YOLO will be able to detect them, it is necessary to prove that.

The model weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

Both the *Dataset 1* and *Dataset 2* were used to train the model. For testing, the testing dataset of *Dataset 1* will be used.

No augmentations were used for first experiment.

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. When using the *Dataset 1* a total of 1000 epochs were run, whereas when using the *Dataset 2* a total of 113 epochs were run.

7.4.2 Results

The obtained training and validation losses are the following:

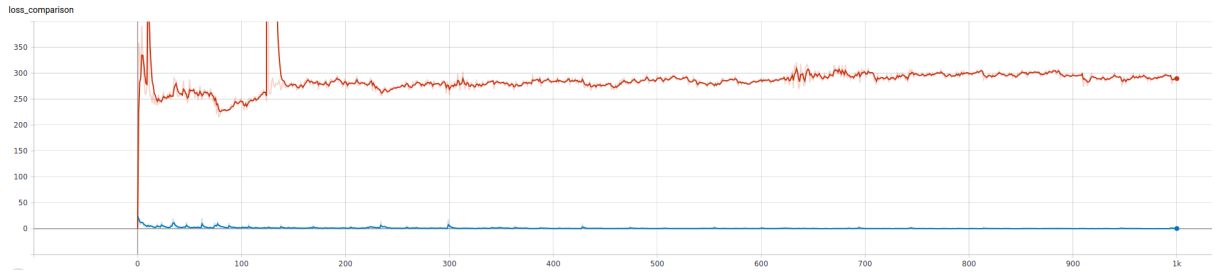


Figure 7.10: Training and validation loss functions when training with *Dataset 1*.

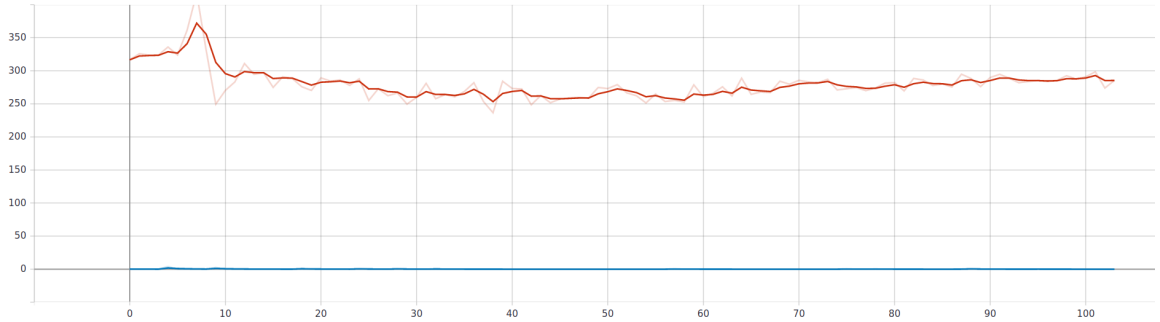


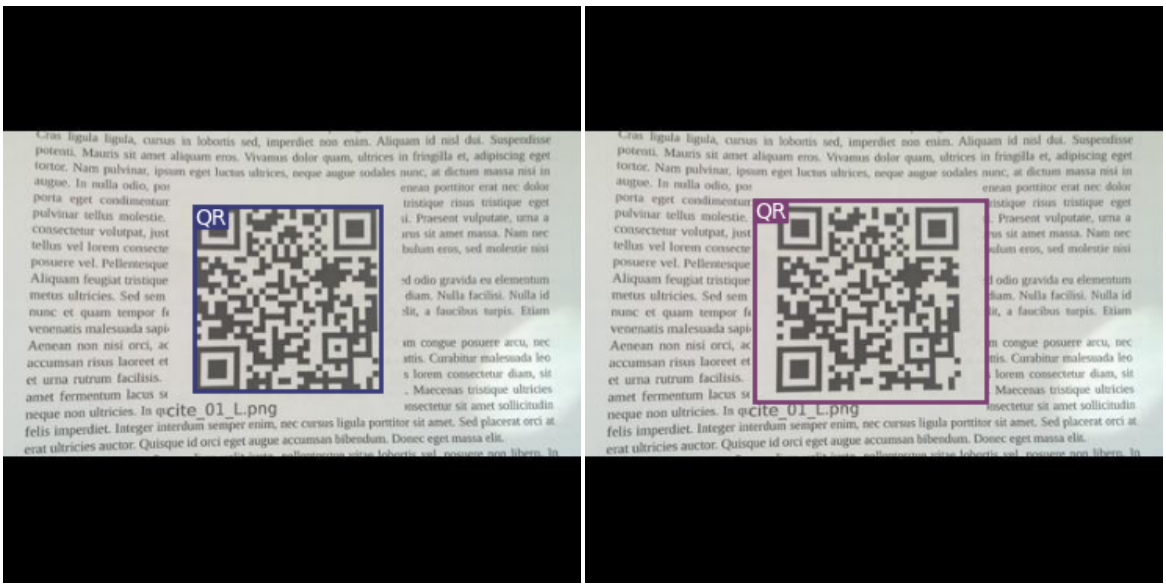
Figure 7.11: Training and validation loss functions when training with *Dataset 2*.

The obtained mAPs are the following:

Metric	Dataset 1	Dataset 2
mAP	0.964	0.759

Table 7.3: Model performance metrics after training with both datasets.

To further visualize how the model performs, the next step is to make inferences in the testing dataset and check the output:



(a) (b)

Figure 7.12: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

(a) (b)

Figure 7.13: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

(a) (b)

Figure 7.14: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

7.4.3 Discussion

When training with *Dataset 1*, the loss comparison seen in figure 7.10 shows that the training loss starts to be reduced at the 20th epoch, and performs as expected until the 120th epoch, in which the validation loss function soars. After this spike, it decreases and remains stable for the rest of the epochs, raising slightly at the end. On the other hand, the loss function in the training dataset decreases quickly during the first 100 epochs, after which it stabilizes and remains near at 0 for the rest of the training.

These results show that the model is training adequately, and that it may start to overfit at the end, when the validation loss starts to increase very slightly. It can be seen, as well, that the small size of the dataset allows for a very sharp decrease in the training loss, since the model is able to learn the features from the few examples available very quickly, and that overfitting is harder to achieve since the training, validation and testing images are very similar between them.

On the other hand, when training with *Dataset 2*, the loss comparison seen in figure 7.11 shows that the model is able to reduce the training loss to almost 0 since the very beginning, and that it keeps it that way for the rest of the training. Contrarily, the validation loss increases during the first 7 epochs and is reduced after that until the 38th epoch, in which it starts to increase very slightly.

It must be recalled that, as virtually all images in this dataset are synthetic and with very similar QR Codes (since their versions range from 1 to 4), the model is able to make detections very easily, since the conditions are optimal (centered QR Code with no noise, white background and no deformation), as expected.

It is very interesting to see how the model trained with the *Dataset 2* has trouble detecting QR Codes (as seen in figures 7.12, 7.13 and 7.14), since some of the times it makes multiple different detections in one QR Code. A possible reason behind this is that *Dataset 2* has a distribution highly biased towards codes of versions 1, 2, 3 and 4, whereas the testing dataset contains more evenly distributed images, which may cause that the model missclassifies as a QR Code a group of alignment patterns (this can be seen in Figure 5).

These results show that detecting QR Codes with *YOLO* is feasible, but the next question that arises is: *¿Could image augmentation be used to improve the obtained results and achieve a model that is able to generalize better?* The next section tries to answer this.

7.5 QR codes with augmentations

7.5.1 Settings

The first batch of experiments main objective was to try to study how the model trains with two datasets with no augmentation.

The augmentations used are the same as the ones explained in section 5.6. They try to generate images with scenes similar to those expected to be found in a real-world scenario. The main objective when using these techniques is to try to increase the performance when working with the testing dataset. This should make validation loss function decrease slower but steadily.

The model weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

Both the *Dataset 1* and *Dataset 2* were used to train the model. For testing, the testing dataset of *Dataset 1* will be used.

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. When using the *Dataset 1* a total of 1000 epochs were run, whereas when using the *Dataset 2* a total of 138 epochs were run.

7.5.2 Results

The obtained training and validation losses are the following:

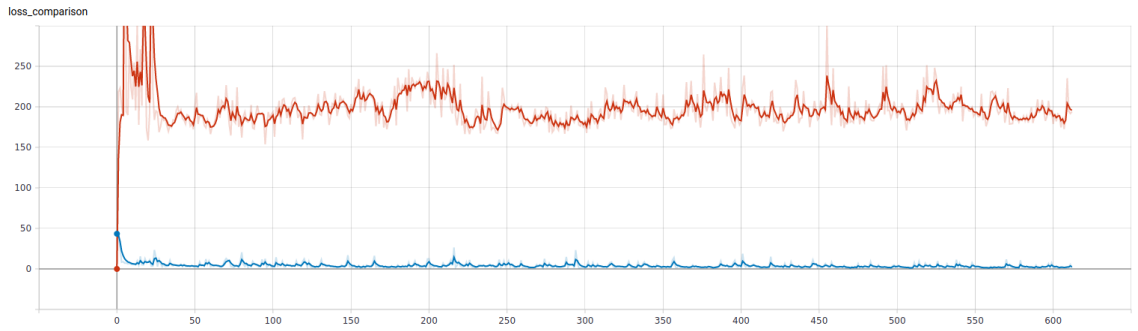


Figure 7.15: Training and validation loss functions when training with *Dataset 1*.

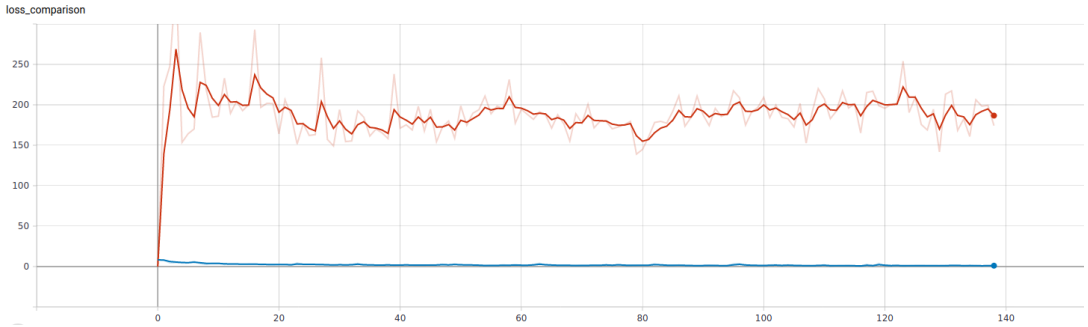


Figure 7.16: Training and validation loss functions when training with *Dataset 2*.

The obtained mAPs are the following:

Metric	**Dataset 1**	**Dataset 2**
mAP	0.887	0.388

Table 7.4: Model performance metrics after training with both datasets and applying image augmentations.

To further visualize how the model performs, seeing some of the inferences with testing images may be of help:

(a) (b)

Figure 7.17: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

(a) (b)

Figure 7.18: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

(a) (b)

Figure 7.19: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*

7.5.3 Discussion

The results show that when training with *Dataset 1*, the training loss is way higher than when not augmenting images (see figures 7.10 and 7.15), and even though it decreases quickly, it stabilizes in a higher value than when no augmentations are applied. The validation loss is more interesting, as it fluctuates a lot during the first 60 epochs, and then stabilizes. This stabilization, though, is not as pronounced as when no augmentations are used. This may be due to the fact that the images are not as stable as before (even though each augmented image is based on the same "base" image, there is a high degree of variation). When stabilized, the validation loss stays at a lower value when using image augmentations, which seems to hint that the obtained results are better.

This situation is very similar in the case of the *Dataset 2*. The training loss starts at a higher value, but then lowers quite rapidly and then stabilizes at almost 0 (see figure 7.16). The validation loss, though, is quite irregular. It decreases very fast, but has 4 high spikes afterwards. Nonetheless, the validation loss is able to decrease until the epoch 78, after which it starts to increase slowly. As in the previous case, the validation loss stays at a lower value when augmenting (see figures 7.11 and 7.16).

In both cases, the observed spikes may be due to the fact that the image augmentation pipeline is introducing changes that cause the network to train with incorrect samples in certain epochs. This especially affects the model when using *Dataset 2*, as seen in the more pronounced spikes in figure 7.16. Since the only difference between the two datasets is the proportion of synthetic images, this may be the reason causing this issue.

It seems that applying image augmentation allows the model to train better, as the validation loss function can be lowered more than when not doing it. A more important question now arises, and is how does this model performs in the testing set. The results seen in figures 7.17, 7.18 and 7.19 show that the results are not as promising as the training and validation losses seemed to tell.

More specifically, the model trained with *Dataset 2* seems to have many issues at performing detections, as many QR codes are incorrectly treated as background. Furthermore, the model trained with *Dataset 1* fails at some predictions that were made correctly (or at least better) when no image augmentations were applied (an example of this can be seen in figures 7.13 and 7.18).

It is worth noting that several iterations of the image augmentation pipeline were done to fine-tune the process until good enough results were obtained. Therefore, it does not seem that this may be the problem. A visual example of the resulting augmented images can be seen in figure 5.1.

Therefore, it seems that both results are worse than when not applying augmentations. This may be explained by the fact that the model needs to train during more epochs to generalize better. Because of the limitations of the hardware available for this project this has not been possible, but this will clearly make one objective for the section 9.

7.6 QR codes with classes

7.6.1 Settings

Now that the QR code detection has been studied, it is time to study the classification of QR codes (i.e. not only detecting a code, but correctly classifying its version).

The classifier weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

For these experiments no image augmentations were performed, and instead of using a single *QR* class, the versions of the QR codes were used. Therefore, model will have to learn to discern between these classes.

For the *Dataset 1*, that meant that there were a total of 12 classes (*QRv1*, *QRv2*, *QRv3*, *QRv4*, *QRv5*, *QRv6*, *QRv7*, *QRv8*, *QRv9*, *QRv11*, *QRv15* and *QRv16*). For the *Dataset 2*, it meant that there were a total of 12 classes as well (the same ones as in the previous case since, as it can be deduced from the explanation of its generation in section 5).

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. When using the *Dataset 1*, a total of 1000 epochs were run, whereas when using the *Dataset 2*, a total of 120 epochs were run.

Considering that this task is harder than just detecting a QR code, a decrease in the mAP is expected. The distribution of QR codes versions of both datasets, explained in section 5, must be taken into account for these experiments.

7.6.2 Results

The obtained training and validation losses are the following:

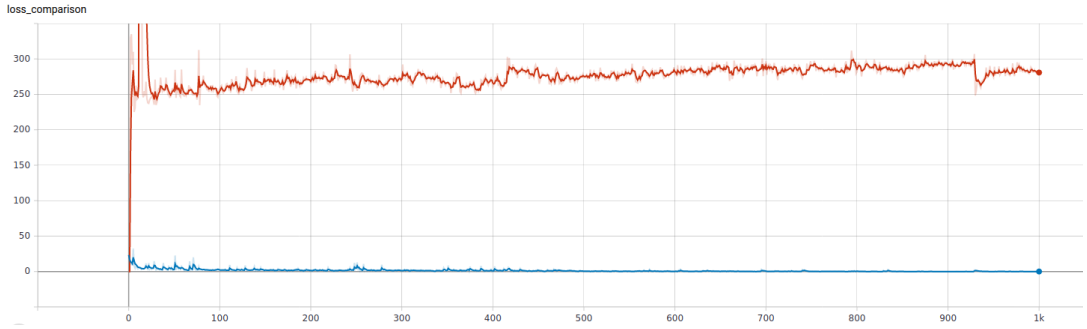


Figure 7.20: Training and validation loss functions when training with *Dataset 1*.

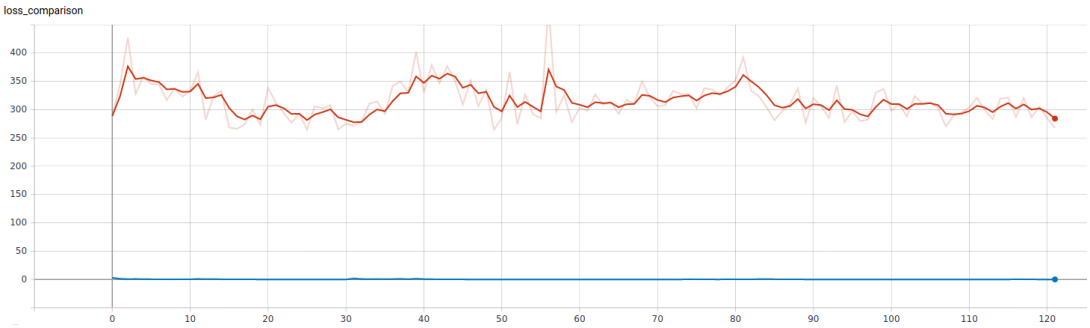


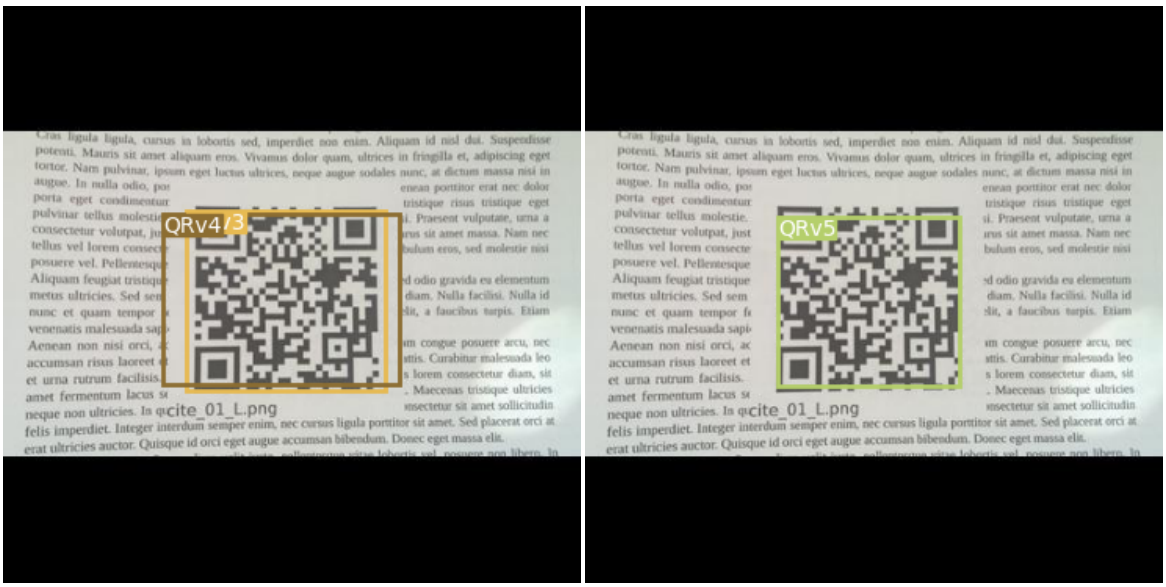
Figure 7.21: Training and validation loss functions when training with *Dataset 2*.

The obtained mAPs were the following:

Metric	**Dataset 1**	**Dataset 2**
mAP	0.462	0.454

Table 7.5: Model performance metrics after training with both datasets.

Again, to better visualize the performance, let's see some of the inferences performed by both models:



(a) (b)

Figure 7.22: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*. The QR version is 3.

(a) (b)

Figure 7.23: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*. The correct QR version is 4.

(a) (b)

Figure 7.24: (a) Detection by model trained with *Dataset 1* (b) Detection by model trained with *Dataset 2*. The correct QR version is 9.

7.6.3 Discussion

The training with the *Dataset 1* seems to be very stable (see figure 7.20), as both the training and validation error stabilize rapidly. It is interesting to note, though, that there is a descent in the validation loss pass the epoch 930.

On the other hand, training the model with the *Dataset 2* is way more unstable (see figure 7.21), with successive increases and decreases in the validation loss.

As expected, the mAP has lowered, since the task is much harder than just locating a QR code. It is interesting to see how training with both datasets provide similar results, even though the obtained loss functions (see figures 7.20 and 7.21) show very different patterns. The results, then, will probably be explained by the fact that the images in the testing dataset do not contain many QR codes with versions ranging from 1 to 4, which are the ones that are added to *Dataset 1* to create *Dataset 2* (an in depth explanation of this can be read in section 5).

From the three predictions (see figures 7.22, 7.23 and 7.24), it can be seen that both networks struggle with the correct version, as they predict several versions for each QR code and, moreover, sometimes none of these predictions is correct (figure 7.24).

As explained previously, the case of figure 7.24 is expected for the model trained with *Dataset 2*, as the extra 10.000 samples contain QR codes with versions ranging from 1 to 4 and, therefore, no increase in the performance for the rest of the QR versions is expected (as is the case with this image).

Here, it is very clear that a better dataset, with a balanced distribution and many more examples, is needed to be able to conclude whether *YOLO* is able of predicting the QR code version or not. Nonetheless, and considering the two datasets with which the experiments have been performed, the results are positive.

7.7 QR codes with classes in a balanced dataset

7.7.1 Settings

The issues of not having big enough datasets has been seen in the previous section. Nonetheless, there is a way to check if *YOLO* is able to correctly classify different versions of QR codes.

To do this, a completely synthetic dataset has been generated: *Dataset 3*. The idea behind it is to create a dataset with the best possible conditions: all the backgrounds are white, the QR have a 0^o orientation, the direction from which they are being pointed at is orthogonal to the QR code plane, and no artifacts are present in the image (no blur or excessive lighting, etc.).

This dataset will be used to train *YOLO* without applying image augmentations, and then inferences will be made over the testing dataset (which will also be synthetic and will have the same conditions mentioned before). If the model is able to correctly distinguish them in this scenario, it will probably make no sense to try to train with images in worse scenarios (blurred images, perspective deformations, etc.).
Therefore, the results obtained in this section should be treated as a baseline of how *YOLO* performs in the classification of different QR versions.

The model weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. The model was trained during 168 epochs.

7.7.2 Results

The obtained training and validation losses were the following:

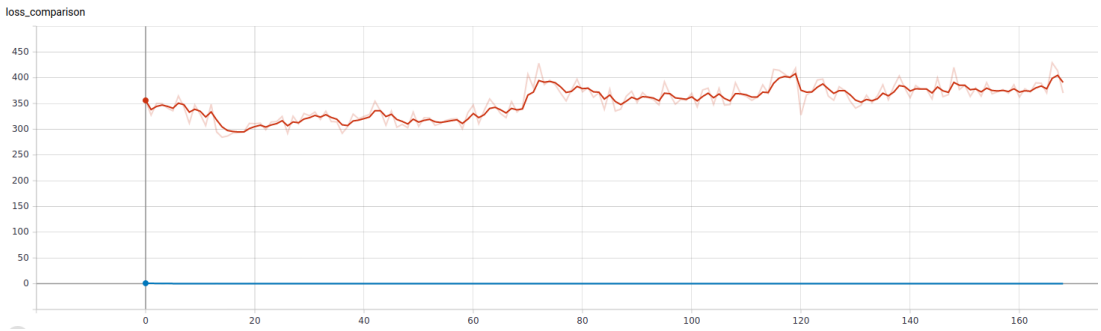


Figure 7.25: Training and validation loss functions when training with *Dataset 3*.

In the testing dataset, the model obtains the following performance:

Metric	Dataset 3
mAP	0.9997

Table 7.6: Model performance metrics after training with *Dataset 3*.

7.7.3 Discussion

As it can be seen in figure 7.25, the training loss is stable at almost 0. The validation loss decreases very quickly and reaches its minimum at epochs 17 and 18 (it has the same value at both of them). After that, it starts to slowly increase and, even though it has some descents, they do not offset the previous increases. This may indicate that the model starts to overfit at these epochs.

The results show that *YOLO* is indeed able of correctly identifying different QR codes. It must be noted, again, that this dataset may not be used in a real case scenario, as it is composed by centered QR codes, with different size (as each QR version differs in size because it contains more information) and with a constant white background.

Nonetheless, as in *Dataset 3* the QR codes are centered and each QR code version has a different size, a question arises after seeing these results: *"Does YOLO really learns how to distinguish between QR code versions, or does it simply classifies them depending on their size?"*. To be able to really conclude that *YOLO* is indeed able of correctly classifying between QR code classes, this question will be answered in the next section.

7.8 Varying bounding boxes sizes

7.8.1 Settings

The next step to check whether *YOLO* is really able to distinguish between the QR code versions is to scale the images, so that the sizes of the bounding boxes are not fixed for each of the classes (see section 5.6.2. If the model is still able to distinguish them with varying bounding boxes sizes, then it could be concluded that it really is able to perform this task.

For this section, the same dataset used in section 7.7 will be used (*Dataset 3*).

The augmentations used in this section to achieve the bounding resizing are defined in section 5.6.2.

The model weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. The model was trained during 158 epochs.

7.8.2 Results

The obtained training and validation losses are the following:

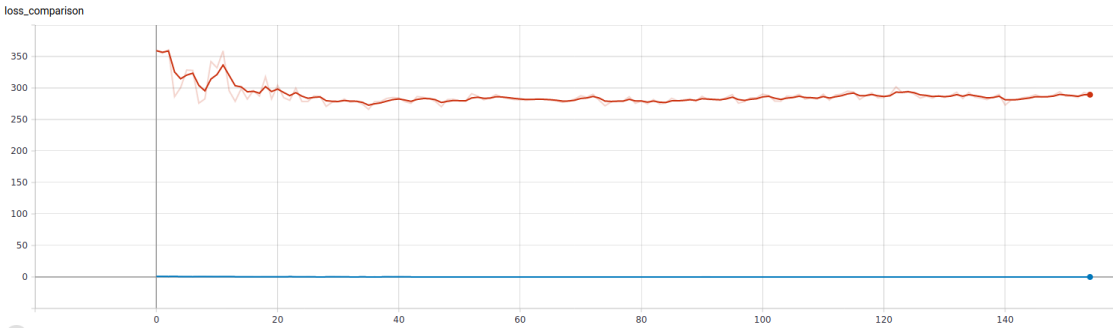


Figure 7.26: Training and validation loss functions when training with *Dataset 3*.

The obtained mAP is the following:

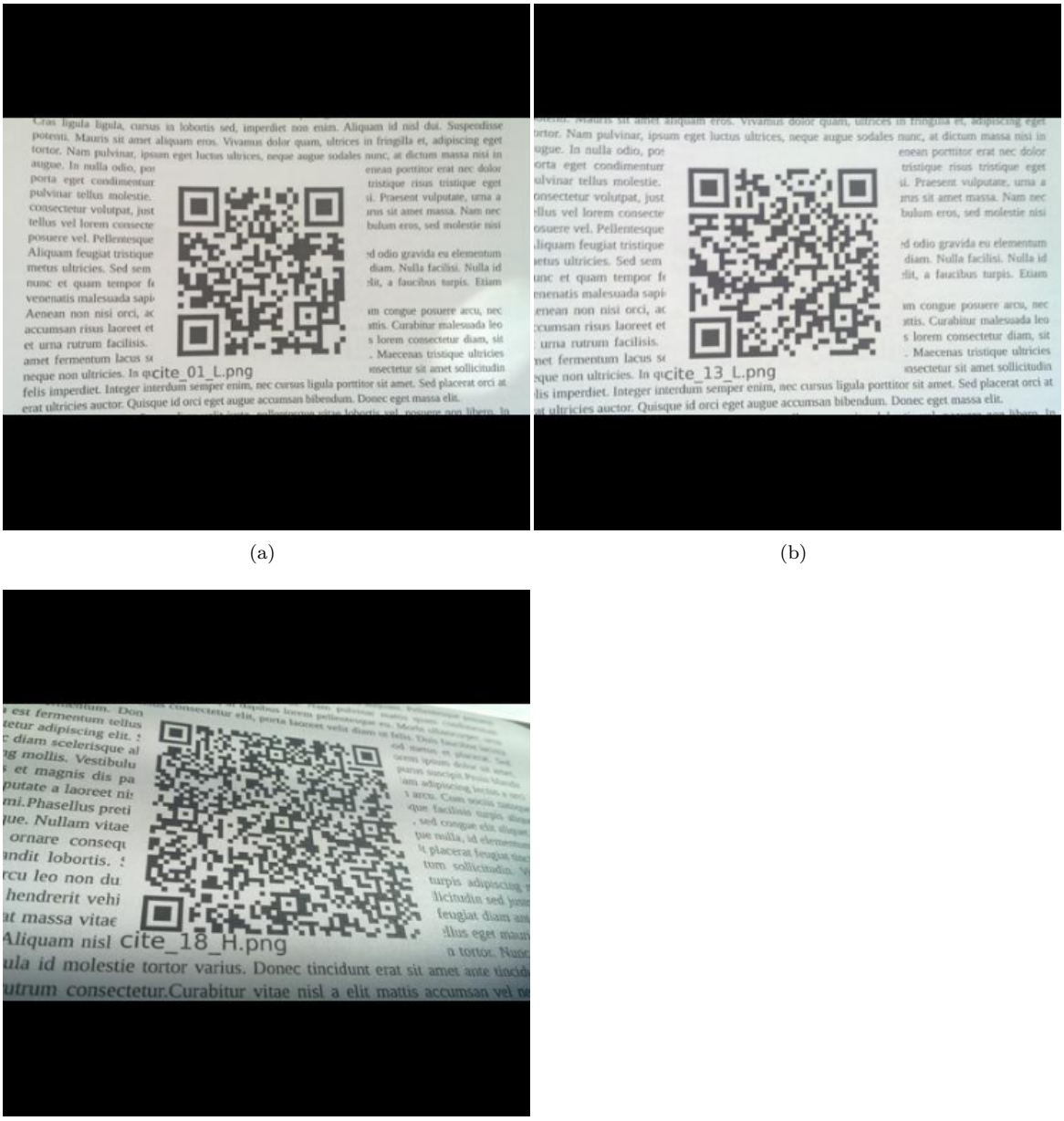
Metric	Dataset 3
mAP	1

Table 7.7: Model performance metrics after training with *Dataset 3* applying the augmentations in section 7.8.

7.8.3 Discussion

These results confirm that *YOLO* is able of correctly differentiating the versions of QR codes, as the obtained performance is not due to the fixed size of each QR version but to its shape.

It is interesting to see how this model performs in detecting the classes of the QR codes of the *Dataset 1* testing dataset. Unsurprisingly, no detections are performed in any of these images (in fact, it does not detect anything in any image of that different testing dataset):



(a) (b)

(c)

Figure 7.27: (a) (b) (c) Detections by model trained with *Dataset 3*.

It is clear that training with a synthetic dataset will not allow the model to generalize and, therefore, a good performance in a real production scenario cannot be expected. This highlights the importance of having a quality dataset with images similar to those expected to be found in the production environment.

7.9 Tiny YOLO

7.9.1 Settings

As stated in section 1.5, one of the initial objectives of the project was to get a model able to perform detections as fast as possible. *Darknet 19*, a shallower version of *YOLO*, seems to be perfect for this. task.

The structure of *Darknet 19* (or *YOLO-tiny*, a variation of the original *Darknet 53*, can be useful for that purpose. It has 19 convolutional layers (which gives its name) and 5 maxpooling layers.

Being shallower than *Darknet 53* its performance may be worse, but it could be still good enough for the required task, with the obvious advantage of the increased speed both in training and when making inferences.

A first run will be executed in which no image augmentations will be applied. In a second run, the image augmentations explained in section 7.8.1 will be used.

The model weights are initialized with values drawn from a normal distribution with mean 0 and standard deviation 0.02.

To make the predictions after training, the weights of the epoch with the highest validation mAP score are used. The model was trained during 140 epochs (when no image augmentations were applied) and 144 epochs (when image augmentations were applied).

7.9.2 Results

The obtained training and validation losses are the following:

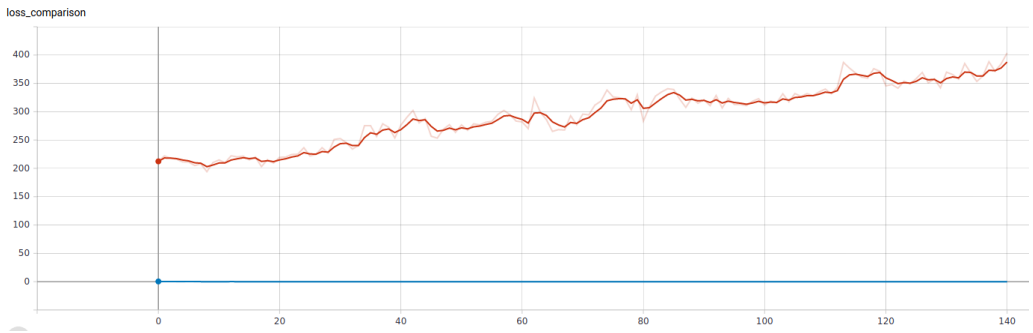


Figure 7.28: Training and validation loss functions when training with *Dataset 3*, using *Darknet 19* and not applying image augmentations.

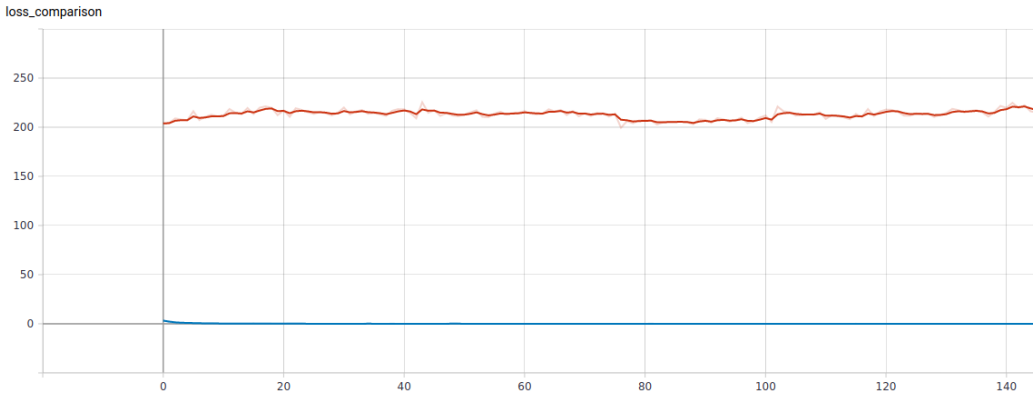


Figure 7.29: Training and validation loss functions when training with *Dataset 3*, using *Darknet 19* and applying image augmentations.

The obtained performance is the following:

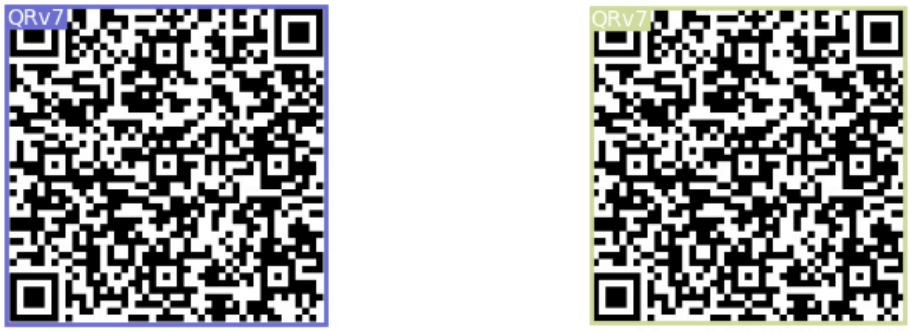
Metric	**Dataset 3**	**Dataset 3 (with augs.)**
mAP	1	1

Table 7.8: Model performance metrics after training with *Dataset 3*, both without applying and applying the same image augmentations as in section 7.8 respectively, and using *Darknet 19*.

Some of the inferences made were these. The predictions shown are made by the model trained with the image augmentations explained in section 7.8 and the model in this section that was trained when applying image augmentations:

(a) (b)

Figure 7.30: (a) Detection by *YOLO* model (b) Detection by *YOLO* model. The correct QR version is 6.



(a) (b)

Figure 7.31: (a) Detection by *YOLO* model (b) Detection by *YOLO* model. The correct QR version is 7.

(a) (b)

Figure 7.32: (a) Detection by *YOLO* model (b) Detection by *YOLO* model. The correct QR version is 7.

7.9.3 Discussion

It is interesting to see how the shapes of the training and validation losses are very similar when using *Darknet 53* (figures 7.25 and 7.26) and *Darknet 19* backbones (figures 7.28 and 7.29), both with and without augmentations respectively.

The reason behind the consistent increase in the validation loss seen in figure 7.28 may be due to the fact that the model is overfitting. This could be possible, but nonetheless the model still manages to get a mAP of 1 in the testing dataset. Since this increase is flattened in figure 7.29 and the only difference between them is the use of image augmentations (see 5.6.2), it may be due to the fact that the model was indeed using the size to classify the QR codes and thus, was overfitting much faster.

In terms of time comparison, the model took 3 hours and 1 minute to train 140 epochs, whereas when using *Darknet 53* it took 9 hours and 28 minutes, which results in a speed-up of 3.14x while keeping the same performance.

The batch size when training both models was set to 4 for the sake of compare the times with equal conditions, but it is noteworthy to say that *Darknet 19* allows for a greater batch size, since the amount of memory that it requires is way lower, which means that the batch size could be increased with the same hardware to obtain an even faster training time.

With 8GB of VRAM (see section 7.2), the total batch size can be increased to 32 and the model still fits in memory (in contrast with the previous 4 batches). Using this batch size allows the model

to be trained during 140 epochs in just 1 hour and 23 minutes, which represents a speed-up of x6.84 compared to using *Darknet 53*. This is especially interesting for small start-ups and spin-offs like *ColorSensing* that have a limited budget to spend in expensive hardware (such as the professional grade GPUs, which can cost from almost 3.000 euros [52] to 10.000 euros [53] (and even more if the data center solutions are taken into consideration).

Each of the inferences made by *YOLO-tiny* took about 0:00:00.005547, in comparison with the time of 0:00:00.025401 needed by *YOLO*, which represents a speed-up of x4.579 is really impressive considering that the model did not loose any performance in terms of mAP. It must be noted, though, than in a real scenario it would most certainly would, but it still is impressive nonetheless.

The obtained results show that *YOLO-tiny* is indeed a very interesting option that should be explored further when training the model with more realistic datasets.

When moving to the predictions comparison, the results shown in figures 7.30, 7.31 and 7.32 show that the predictions made by both models are virtually the same, which shows that no performance is lost when using *YOLO-tiny*. It must be noted, though, that this would most probably not be the case when training and making predictions in a dataset containing real images instead of synthetic ones.

Chapter 8

Conclusions

Along all the experiments performed in section 7, the different hypothesis stated in section 7 have been proved. Let's recall them and how have they been confirmed:

1. **Can transfer learning be used to perform QR codes detection?** This has been checked in section 7.3, and it has been seen that transfer learning indeed gives a good performance. So good, indeed, that it would be very interesting to repeat all the experiments using it to see how the performance varies.

2. **Can *YOLO* detect QR codes?** Indeed it can. Sections 7.3, 7.4 and 7.5 prove that *YOLO* is able of detecting QR codes. The results also show that the dataset plays a central role in the performance that the model is able to yield. But the results, nonetheless, are good enough to confirm this hypothesis.

3. **Can image augmentations be used in combination of synthetic images?** This has been proved in section 7.8, but the results obtained in section 7.5 have not been so positive. This should be further study to analyze whether if the decrease in performance seen in section 7.5 is due to a lack of training or to a poor image augmentation pipeline. The second option seems unlikely, for the reasons explained in section 7.5.3.

4. **Can *YOLO* classify QR codes by their version?** Indeed it can. This has been proved in sections 7.6, 7.7 and 7.8. The performance obtained in these sections proves that *YOLO* is more than capable of doing this.

5. **Can a shallower and faster version of *YOLO* be used to classify QR codes by their version without loosing too much performance?** Yes. It has been proved in section 7.9, in which *YOLO-tiny* has been used to classify the synthetic images from *Dataset 3*. The results have been excellent, and encourage further study to show the balance between the loss of performance and the increase in inference speed in a dataset containing real images rather than synthetic ones.

In the sections that follow, the main conclusions drawn out of the experiments will be explained in more detail.

8.1 Weighted cross entropy

As seen in sections 7.6 and 7.7, having a dataset in which the classes are not evenly distributed can be a problem, especially when there is an extreme imbalance, in which case the model may totally ignore the less common classes. A solution to this problem could be to apply weighted cross entropy, which can be used to force the model to weight more the classes which are less frequent.

8.2 Dataset importance

All the subsections of the section **??** lead to a common issue: the need of a proper dataset to train. In section 7.8, it has been seen how a model with a synthetic dataset with balanced classes is not enough to make predictions on a testing dataset containing real images. This is obvious, as the distributions of both datasets are not the same, as one contains synthetic images with "perfect conditions" (white background, no deformations, orthogonal view angle with respect to the QR code) whereas the other contains actual images, with what that entails (deformed QR codes, blurred pictures, perspective deformations, etc.).

A typical technique used to try to cope with this problem is image augmentation, but as section 7.5 shows, it is not enough when the training and testing datasets distributions are too different.

Having a large dataset, though, is not easy. The three datasets generated (section 5) took a greater amount of time than expected, and in situations like this project (with a time-span limited to three months of work), generating and labeling a dataset large enough to obtain a good performance would probably take more time than actually performing experiments with it.

8.3 QR code detection

The main task of detecting QR codes has been performed with mixed results. Section 7.4 has proved that it is indeed possible, but section 7.5 has shown poor results, especially when working with *Dataset 2*. It was expected that the image augmentation process improved the results or, at least, did not make them that much worse. This may indicate that some improvement in the image augmentation process could be achieved, such as reducing the effects to deform the images to a lesser extent.

Furthermore, the results obtained in section 7.3 prove that the model is indeed able of performing the detections correctly, and it encourages the future work of deeply analyzing how transfer learning would perform in the experiments.

8.4 Transfer learning

As seen in section 7.3, transfer learning seemed to provide some interesting results. Even though the validation loss was worse than when not using the pretrained *Darknet 53* backbone, the performance in terms of mAP was indeed very good (and, when using *Dataset 2*, it outperformed the model trained from scratch).

Even though it has not been done because due to time constraints, repeating the experiments applying transfer learning could provide very useful insights and would very probably yield slightly better results than those obtained in some of the experiments.

8.5 QR version classification

One of the main objectives of this project was to study whether if *YOLO* was able of correctly classifying QR codes versions. In sections 7.6 and 7.7, it has been seen that *YOLO* is indeed able of performing this task for QR versions ranging from 1 to 13 in the best possible conditions (since a synthetic dataset was used). It is expected that the model would be able to do that as well with greater versions of QR codes.

This conclusion is pretty positive for this project, since it confirms the initial hypothesis that *YOLO* can be used for this task with a high degree of success, but does not confirm that it would perform well enough in a real scenario, as explained previously, the dataset contains synthetic images only with no deformations whatsoever. Therefore, a system implementing this model could have issues detecting QR codes under worse conditions (a heavily deformed QR code, a blurry image, etc.).

The degree up to which this is a problem depends on the task at which the model is used. In some scenarios, such as many of the use cases of *ColorSensing's* technology, not being able to locate a QR code may just require the user to re-upload the photo with better ambient conditions (i.e. better focus and illumination, etc.).

8.6 Inference times

As section 1.5 states, achieving a model that is both accurate and fast is very important. The experiment in section 7.9 tries to study that.

The results seen in section 7.9 show that *YOLO-tiny* is able to perform inferences at a speed of 180 FPS, whereas *YOLO* is able to perform inferences at a speed of 39 FPS, which proves that both systems are able of achieving real-time speeds (considering as real-time everything above 30 FPS, using the same criteria as in [29] or [18]).

Furthermore, it has not been seen any decrease in performance in terms of mAP when switching to *YOLO-tiny*, which is a good signal, considering that the tests were performed in a dataset containing only 9.750 synthetic images. A decrease in this scenario would have been worrying, and these results are really a nice starting point from which further experiments with more complex datasets should be performed.

8.7 Explainability

Modern convolutional neural networks such as the one used in this project, offer an incredible performance as seen by the progress made in recent years [54]. Though the performance is very important for a model, so are transparency and interpretability since these allow to understand why the model is producing the results and to decide if they are correct or not.

However, most neural networks are black boxes that produce results through a decision-making process that it is not easy to understand (and even, it may not be fully understood) [55].

The main reasons why a model could be a black box are because the function is too complex to understand for a human to understand or that the function is proprietary, and therefore its source code is not available for study [56].

In some scenarios, such as an image detector working in non-critical tasks (consider a silly example such as a model that predicts the probability of rain considering the sky in an image) this may not be a grave issue. But now consider a system that decides whether to concede or deny a parole [57]; in this situation, it is essential to know exactly why the model made that specific decision to check if it could have made any mistake or if it could have some kind of racial bias, for example.

A new trend has been to build a second model that tries to explain the decision made by the first model, but this is problematic, as explanations are often not reliable [56].

It is obvious that this project will not solve this issue, and therefore this problem is seen here as well. The discussions of some of the experiments (an example of this can be seen in section 7.5.3) have been quite difficult, as the reason behind the results cannot be fully understood. Nonetheless, educated guesses could be performed with enough confidence to conclude that the reason behind the results are the explained ones.

This first-hand example of the problem with explainability has been enlightening, and it is clear that a deeper study on the explainability of *YOLO* would be very interesting.

8.8 QR code segmentation

The segmentation of QR codes was one of the main objectives of this project, but it has been ruled out of the project due to the time constraints.

The generation of the datasets, as explained in section 5, took more time than expected. This, as well as the fact that the labels in a segmentation dataset were totally different, caused a change in the initial prioritization of the objectives: instead of working to perform experiments in both tasks, we would focus in the object detection to analyze it deeply and would rule out the segmentation task. This was deemed better than performing both tasks at the same time, which could cause neither of both could be studied in proper detail.

It is worth to mention that some exploratory work was performed, and it is clear to the author that *YOLACT* [58] would be the chosen model to perform the task, mainly because of the fact that the authors state that *"...our goal is to fill that gap with a fast, one-stage instance segmentation model in the same way that SSD and YOLO fill the gap for object detection."*.

It is apparent, reading the paper, that *YOLACT* is very related to *YOLO* in the eyes of the authors. In the *Appendix* section, the authors even assert that *"Because the improvements to our detection performance in our observation come mostly from using FPN and training with masks (both of which are orthogonal to the improvements that YOLO makes), it is likely that we can combine YOLO and YOLACT to create an even better detector."*.

Using *YOLACT*, therefore, may be even more profitable coming from *YOLO*, as having previous knowledge of both models could come in handy in the future.

Chapter 9

Future work

Along the experiments sections, some ideas and further experimentation emerged. Even though these could not be done due to time constraints, they are interesting enough so that they should be done in a follow-up of this project. The main ideas are explained in the following sections.

9.1 Anchor boxes

The anchor boxes used in *YOLO* have not been modified. As explained in [29], the anchor boxes are chosen by using k-means clustering on the training set bounding boxes to automatically find good priors. This has not been modified for this project, and it would be interesting to perform the same process to select the appropriate number of anchor boxes.

This could improve the model performance both in terms of training and inference time, since (maybe) the number of anchor boxes (and therefore, the total predictions) could be reduced while at the same time using better anchor boxes sizes, which could boost the mAP.

9.2 Architecture modification

As seen in section 7.9, using *YOLO-tiny* provided a similar mAP when training with *Dataset 3*. As has been explained, this most probably is due to the fact that *Dataset 3* uses very simple images.

Nonetheless, it would be very interesting to see whether modifying the model to further reduce its depth could provide a similar performance to *YOLO-tiny* while keeping a similar mAP.

The explanation to this is the fact that both *YOLO* and *YOLO-tiny* have been designed to perform detections in very semantically big datasets, which contain a wide variety of classes. When predicting QR codes though, the number of classes could be reduced to 10 or 20 (since the biggest QR code versions are not very common and *ColorSensing* decides which versions are going to be used for the printed labels, those not used could be ignored altogether). Therefore, the feature extractor's depth maybe could be further reduced while keeping a good performance.

9.3 QR code segmentation

As explained in section 8.8, the segmentation of QR codes has been ruled out of the project due to time constraints. Since being able of performing this task is still interesting to *ColorSensing*, a pretty obvious future work is to generate the necessary labels for *Dataset 1*, *Dataset 2* and *Dataset 3* to use them in this task.

As explained in section 8.8, *YOLACT* was the model that was researched to be used in the segmentation task, and it seems a perfect match. With the proper amount time, a more thoroughly analysis of the segmentation models could be done to confirm or reject this hypothesis, and proceed to study the QR code segmentation with the chosen model.

9.4 Better datasets

As is explained in section 5, the datasets consists mostly of synthetic images due to the issue of not being able to obtain the expected images to build the originally planned dataset with real images.

This has affected the project, because the experiments that would have proved the feasibility of *YOLO* in a more production-like datasets could not be performed. Therefore, this task still is pending, and it is still needed to validate the obtained results.

9.5 Hyper-parameter tuning

Hyper-parameters tuning has not been properly studied in this project.

Varying the hyper-parameters defined in section 7.2 could be interesting, specially modifying the learning rate to see how it affects the learning time.

For example, a nice idea presented in [18] is the use of genetic algorithms to select optimal hyper-parameters. This could be applied here as well to select the appropriate ones.

A variation in the optimizers used would be interesting as well. As explained in section 7.2, Adam has been used as the optimizer for all the experiments. Using different optimizers could help in the training process.

9.6 YOLOv4

On April 23rd, Bochkovskiy et. al published the 4th version of *YOLO* [18]. This new version provides an increase in mAP of about 10% in the MS COCO dataset:

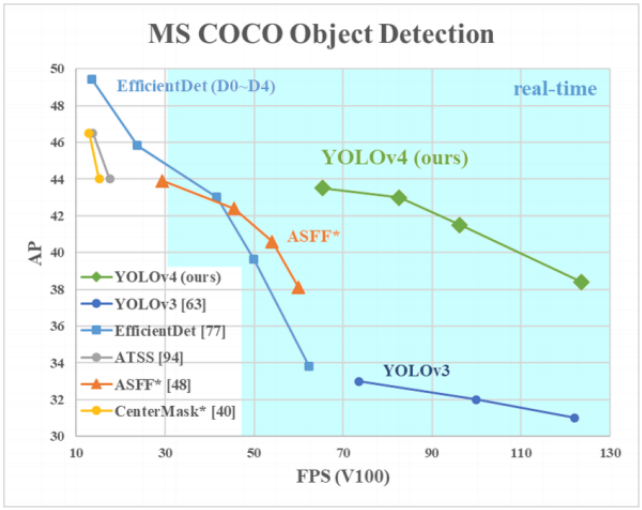


Figure 9.1: Improvement achieved by *YOLOv4* (Source: [18])

The main novelties that this new version introduces are the following:

1. **Bag of Freebies**: BoF are improvements that only increase the training cost to obtain better performance like, for example, image augmentation. The authors discuss several methos, and choose some specific ones like CutMix or DropBlock regularization, among others.

2. **Bag of Specials**: BoS are plugin modules and post-processing methods that only increase the inference cost to obtain better performance like, for example, the SPP module. The authors, among others, choose Mish activation and Cross-stage partial connections.

3. **Architecture neck**: The authors introduce a block between the backbone, which is *CSP-Darknet 53* and the head, which is *YOLOv3*. The neck is SPP (Spatial Pyramid Pooling) and PAN (Path Aggregation Network).

4. **Mosaic augmentation**: The authors also introduce a new augmentation method called Mosaic. It combines 4 images of the training dataset into 1 image.

Using this new *YOLO* version could be interesting to see how the performance varies when dealing with QR codes.

Chapter 10

Acknowledgements

I would like to thank Javier Ruiz-Hidalgo, who directed this project and provided me with incredibly useful insights over many topics, and without whose help this project would not have been possible.

I would also like to thank Oscar Romero Moral, who acted as a speaker for this project and provided many comments which helped to improve this thesis.

I also would like to thank Ismael Benito Altamirano, who acted as the company director for this project.

I would like to mention Daniel Prades García, who provided very interesting ideas to experiment with.

Lastly, but most importantly, I would like to thank my mother all the support and encouragement that she gave me throughout the realization of this thesis.

I deeply hope this thesis to be worthy of all of them.

Bibliography

[1] Colorsensing. `https://www.color-sensing.com`. Accessed: 2020-04-27.

[2] ADC Denso. Qr code essentials, 2011. *URL http://www. nacs. org/LinkClick. aspx*.

[3] International Organization for Standardization. Information technology — automatic identification and data capture techniques — bar code symbology — qr code. *https://www.iso.org/standard/30789.html*.

[4] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[5] Lawrence G Roberts. *Machine perception of three-dimensional solids*. PhD thesis, Massachusetts Institute of Technology, 1963.

[6] Seymour A Papert. The summer vision project. 1966.

[7] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[8] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Geoffrey Hinton, Simon Osindero, Max Welling, and Yee-Whye Teh. Unsupervised discovery of nonlinear structure using contrastive backpropagation. *Cognitive science*, 30(4):725–731, 2006.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Zxing. `https://github.com/zxing/zxing`. Accessed: 2020-05-03.

[14] pyzbar. `https://github.com/NaturalHistoryMuseum/pyzbar/`. Accessed: 2020-05-03.

[15] Detection and segmentation through convnets. https://towardsdatascience.com/detection-and-segmentation-through-convnets-47aa42de27ea. Accessed: 2020-06-17.

[16] Tamás Grósz, Péter Bodnár, László Tóth, and László G Nyúl. Qr code localization using deep neural networks. In *2014 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE, 2014.

[17] Tzu-Han Chou, Chuan-Sheng Ho, and Yan-Fu Kuo. Qr code detection using convolutional neural networks. In *2015 International conference on advanced robotics and intelligent systems (ARIS)*, pages 1–5. IEEE, 2015.

[18] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[19] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact++: Better real-time instance segmentation. *arXiv preprint arXiv:1912.06218*, 2019.

[20] Luiz Belussi and Nina Hirata. Fast qr code detection in arbitrarily acquired images. In *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, pages 281–288. IEEE, 2011.

[21] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[22] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[23] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[24] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[25] Deep object detectors. https://www.slideshare.net/IldooKim/deep-object-detectors-1-20166. Accessed: 2020-05-14.

[26] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[27] Daniel Kold Hansen, Kamal Nasrollahi, Christoffer B Rasmussen, and Thomas B Moeslund. Real-time barcode detection and classification using deep learning. In *IJCCI*, pages 321–327, 2017.

[28] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[29] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[30] Imagenet. https://pjreddie.com/darknet/imagenet/. Accessed: 2020-05-09.

[31] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] Qi-Chao Mao, Hong-Mei Sun, Yan-Bo Liu, and Rui-Sheng Jia. Mini-yolov3: Real-time object detector for embedded applications. *IEEE Access*, 7:133529–133538, 2019.

[34] Bakri Badawi, Teh Noranis Mohd Aris, Norwati Mustapha, and Noridayu Manshor. Color qr code recognition utilizing neural network and fuzzy logic techniques. *Journal of Theoretical & Applied Information Technology*, 95(15), 2017.

[35] Jennifer C Molloy. The open knowledge foundation: open data means better science. *PLoS biology*, 9(12), 2011.

[36] Fira d'empreses ub. http://www.ub.edu/fisica/firaempreses/. Accessed: 2020-06-09.

[37] Qr dataset. http://www.fit.vutbr.cz/research/groups/graph/pclines/pub_page.php?id=2012-JRTIP-MatrixCode. Accessed: 2020-05-09.

[38] Numpy random choice. https://numpy.org/devdocs/reference/random/generated/numpy.random.choice.html. Accessed: 2020-05-15.

[39] Synthetic qr dataset. https://www.kaggle.com/coledie/qr-codes. Accessed: 2020-05-09.

[40] imgaug. https://github.com/aleju/imgaug. Accessed: 2020-05-29.

[41] torchvision. https://pytorch.org/docs/stable/torchvision/index.html. Accessed: 2020-05-29.

[42] Pytorch. https://pytorch.org. Accessed: 2020-05-29.

[43] Pytorch-yolov3. https://github.com/eriklindernoren/PyTorch-YOLOv3. Accessed: 2020-05-15.

[44] Tensorboard. https://tensorflow.org/tensorboard. Accessed: 2020-05-29.

[45] Tensorflow. https://tensorflow.org. Accessed: 2020-05-29.

[46] fast.ai embracing swift for deep learning. https://www.fast.ai/2019/03/06/fastai-swift/. Accessed: 2020-06-19.

[47] Intersection over union. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. Accessed: 2020-05-15.

[48] Adam optimizer. https://pytorch.org/docs/stable/optim.html#torch.optim.Adam. Accessed: 2020-05-09.

[49] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[50] Darknet 53 imagenet weights. https://pjreddie.com/media/files/darknet53.conv.74. Accessed: 2020-06-06.

[51] How to use learning curves to diagnose machine learning model performance. https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/. Accessed: 2020-06-19.

[52] Nvidia rtx titan. https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/. Accessed: 2020-06-08.

[53] Nvidia tesla v100. https://www.flytech.es/producto/tesla-v100/. Accessed: 2020-06-08.

[54] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[55] Zebin Yang, Aijun Zhang, and Agus Sudjianto. Enhancing explainability of neural networks through architecture constraints. *arXiv preprint arXiv:1901.03838*, 2019.

[56] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[57] When a computer program keeps you in jail: how computers are harming criminal justice. new york times. https://www.nytimes.com/2017/06/13/opinion/how-computers-are-harming-criminal-justice.html. Accessed: 2020-06-16.

[58] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: real-time instance segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9157–9166, 2019.